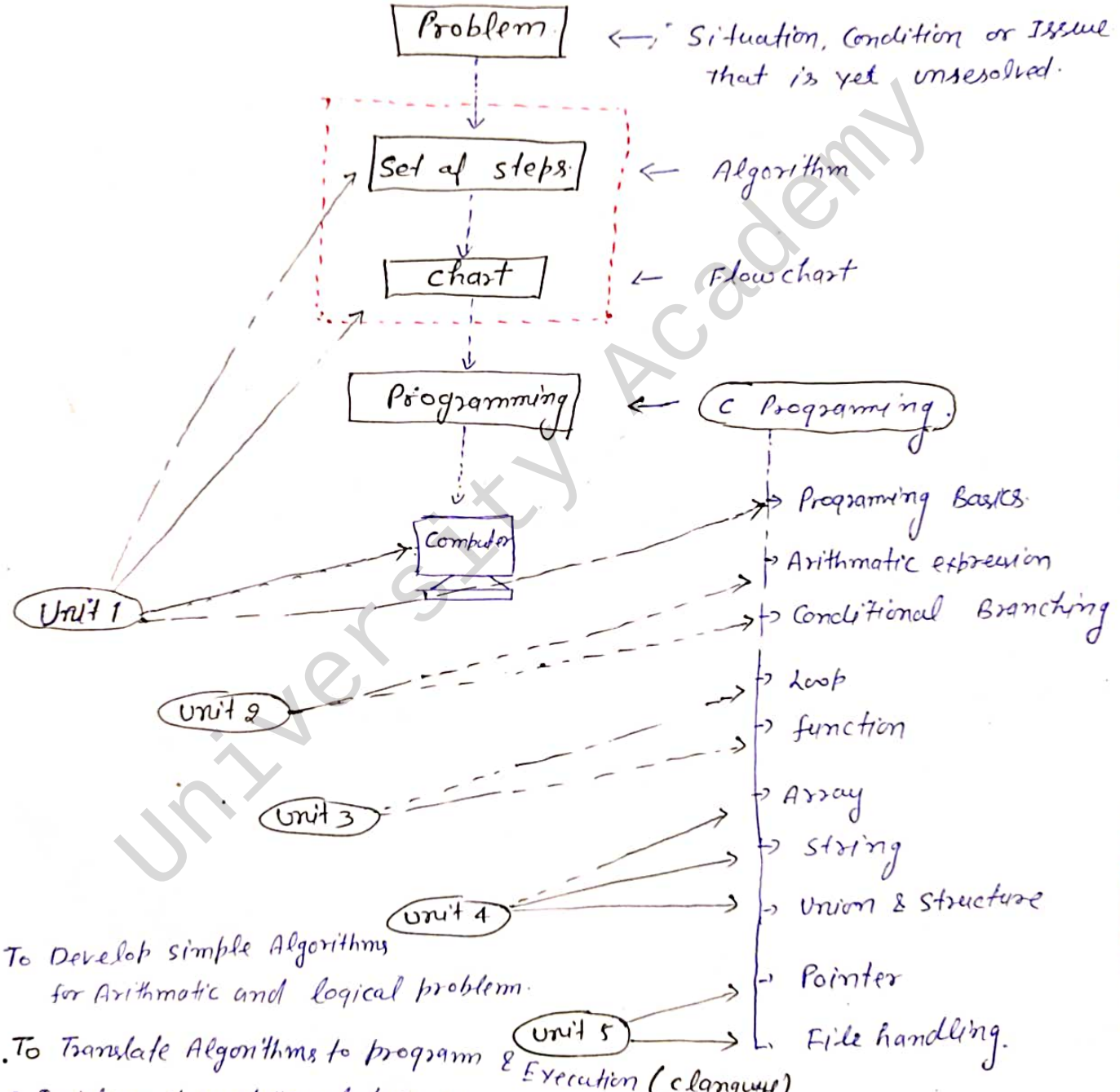
PROGRAMMING FOR PROBLEM SOLVING

Overview of Subject



CO1: To Develop simple Algorithms for Arithmatic and logical problem.

CO2: To Translate Algorithms to program & Execution (clanguage)

CO3: To Implement conditional branching, Iteration & Recursion

CO4: To decompass a problem into functions and synthesize a complete program using devide & conquer approach.

CO5: To Use Array, pointer, and structure to develop algo and program.

UNIT - 1

1.1 What is Computer ?

Computer is a programmable electronic device designed to solve different problem, process data, store and retrieve data, perform faster & efficiently than human.

The term computer is derived from Latin word "Computare" this mean to calculate, to count, to sum up, or to think together.

History of Computer

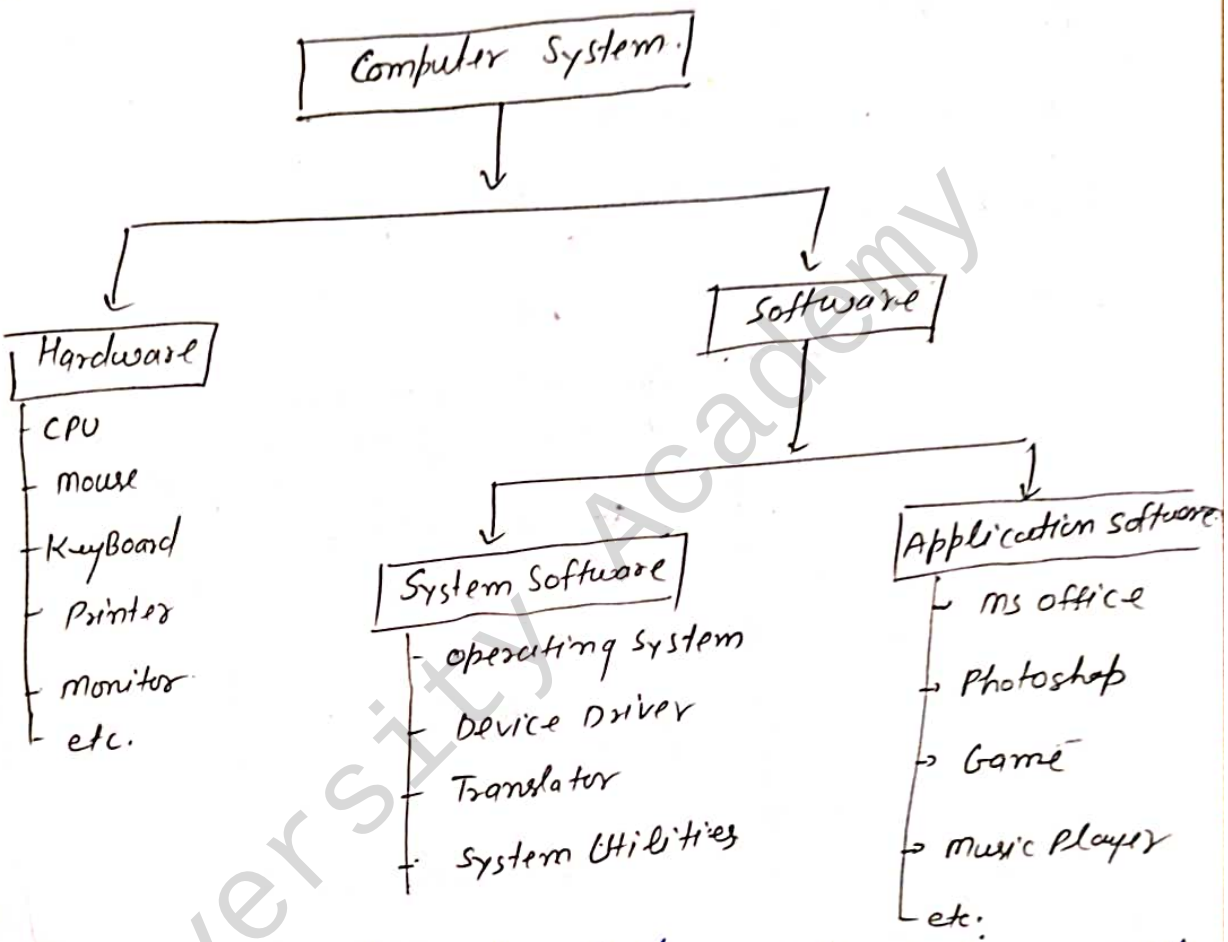
1. ABACUS : To count number, invented around 4000 years ago.
2. Napier's Bone : To multiply and divide invented in 1617.
3. Slide rule : To perform Addition, Subtraction, multiplication and division, invented in 17th century.
4. Pascal's Machine : To perform Addition and subtraction, invented in 17th century.
5. Leibniz's Machine : To perform multiplication & division, invented in 17th century.
6. Computer : Charles Babbage built Mechanical machine to do complex calculation. in 1833.

Generation of Computer

Generation	Technology (chips)	Language.	speed (time)	Size (Infrastructure)	Example.
First Generation (1940-1956)	Vacuum Tube	Machine Language (0,1)	Milliseconds	Large room	UNIVAC, ENIAC
Second Generation (1957-1963)	Transistors	Assembly Language	microsecond	Reduced size	PDP-8 IBM 1401
Third Generation (1964-1971)	Integrated circuits.	High level Language (eg. C, C++, JAVA)	nanosecond	Quite small	IBM 370, PDP 11.
Fourth Generation (1972-1980)	Microprocessors VLSI (very Large Scale Integration	High level Language	Picoseconds	Personal computer: Small size	IBM, Apple,
Fifth Generation (1980- Present)	Artificial Intelligence Super large scale Integrated (SLSI)	High level Language, Machine learning	femto second or faster.	very small	Expert System NLP, Speech Reco; Supercomputer

1.2 Introduction to Components of a Computer System.

A Computer system Consist two major component Hardware and software;



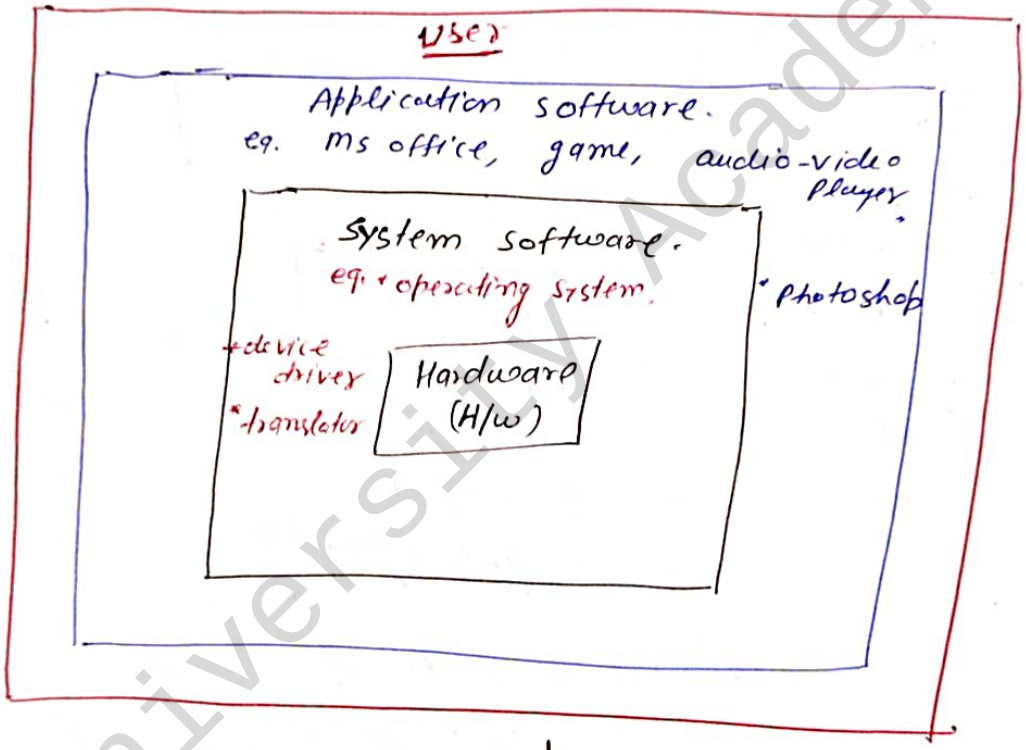
Hardware: The Physical component of computer system which can Interconnected called Hardware. we can see and touch it. eg: CPU, Mouse Keyboard, Printer, Monitor etc.

Software: The set of Instruction to perform any operation is called Program and the collection of program is called software. the computer cannot perform any operation without software.

There are two type of software : System s/w and Application s/w.

System Software: System Software is set of programs that control and manage the operation of computer hardware, it also helps Application program to execute correctly.

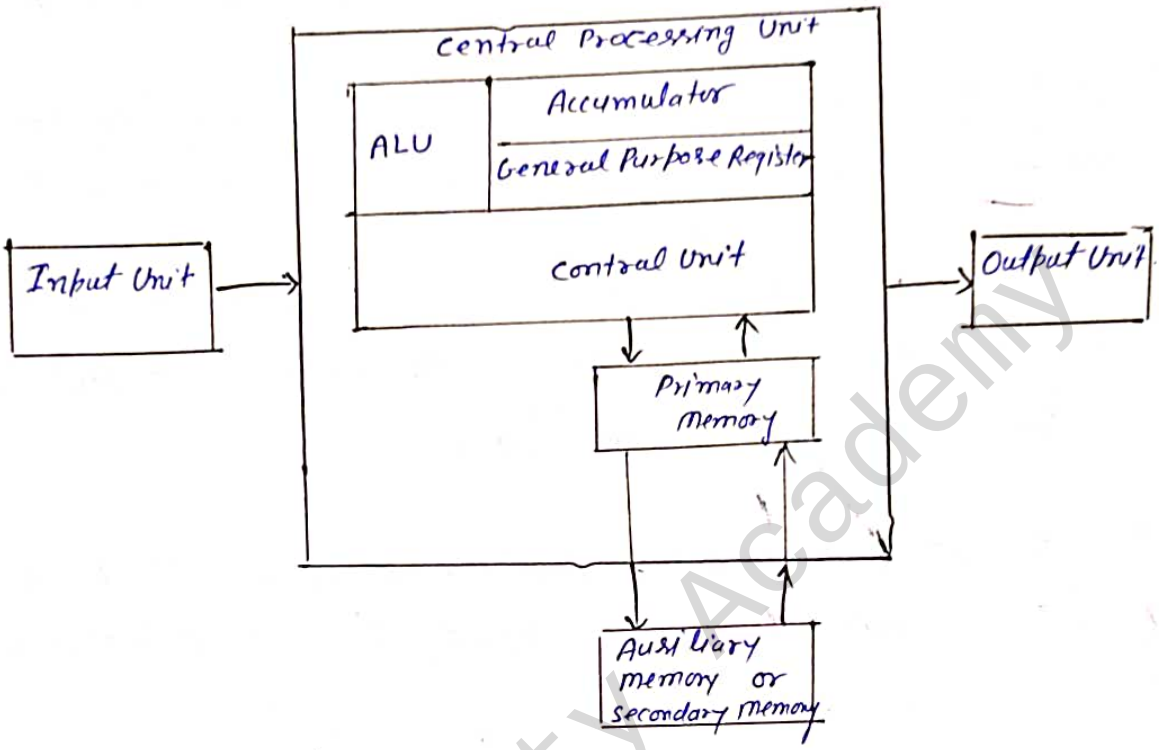
Application Software: Application software is a program that does real work for user. It mostly created to perform specific task for a user.



System software Vs Application software.

System software	Application software
1. System s/w manage the resource of computer like memory, process, security.	1. Application s/w full fil the requirement of user to perform specific task.
2. Written in low level like Machine or Assembly language.	2. A High level language used to write Application s/w.
3. it is general purpose software	3. It is specific purpose software.
4. System s/w are independent of Application s/w.	4. Application s/w need system s/w to run.
5. Example: OS, Compiler, driver etc.	5. Example: ms office, web browser, mediaplayer

1.2.1. Block Diagram of Computer System.



Central processing Unit (CPU): It is the Brain of Computer System. All major calculation and comparisons are made inside the CPU. It is also responsible for activation and controlling the operation of other unit.

The CPU Consist of two major component

(i) Arithmetic logic Unit (ALU) and (ii) Control Unit (CU)

Arithmetic logic Unit (ALU): ALU Performs all the arithmatic operation such as addition, Subtraction, multiplication and division and uses logical operations such as AND, OR, NOT etc.

Control Unit (CU): CU Controls all the operation including control of input/output devices and primary memory.

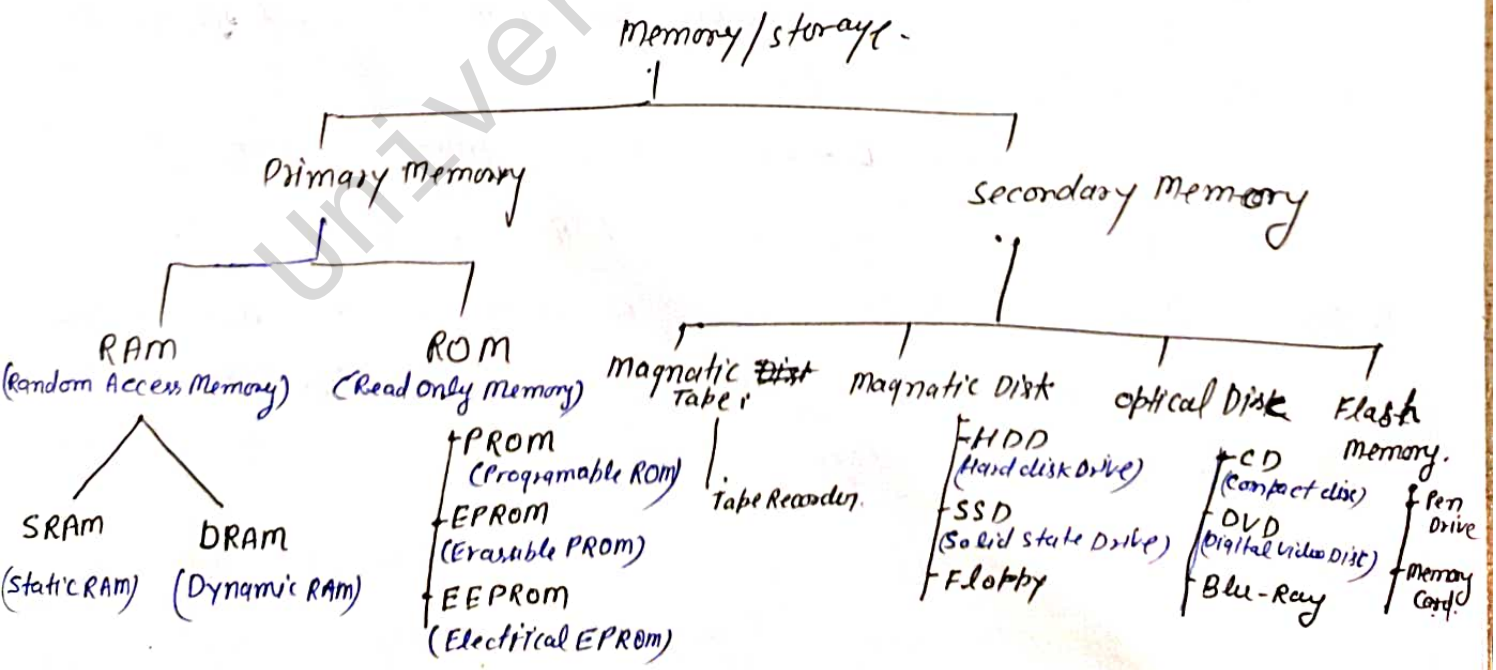
Primary Memory: It is simply known as memory unit. It is essential component of computer. It stores the input data and calculation result. eg. RAM (valatile), ROM (Non-Valatile)

Auxiliary Memory: It is also known as secondary storage. It stores data permanently for long time. eg. Harddisk, CD, DVD.

Input: the user provide the set of instruction or information to the computer system with the help of Input devices. eg. Keyboard, Mouse, Scanner etc.

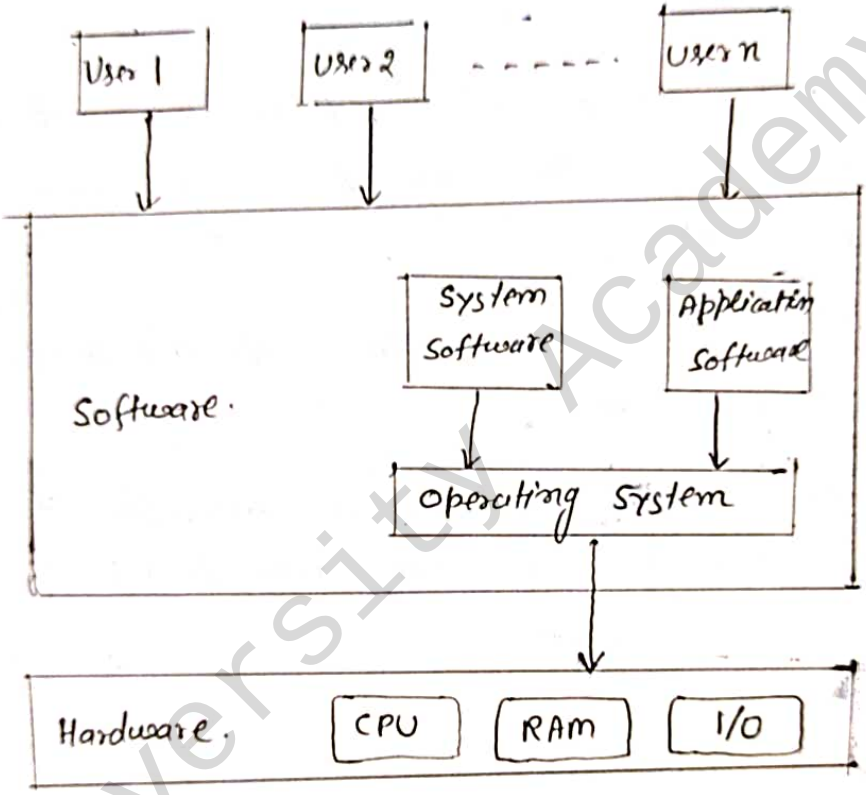
Output: the Output devices produce or generate the desired result according to our input such as printer, monitor etc.

1.2.1 Memory / storage Classification.



1.2.3. Operating System.

An operating system is a program that acts as an interface betⁿ the user and the computer hardware and controls the execution of all kind of programs. Some popular operating systems are. Linux, Windows, MacOS. etc.



Functions of operating System.

i) Memory Management: It refers to management of primary memory and keep track of it. Operation System also do the allocation and de-allocation of the memory.

2) Process Management: All the processes these are given by user or system own process are handled by the operating system.

3) Device Management: Operating System keep track of all devices such as input/output devices. OS also decide which process gets the device when and for how much time.

4) File Management: A file is normally organized into directories. for easy navigation and uses. the directories may contain files and other directories.

5) Security: operating System prevents unauthorized access of program and data by providing user login with password.

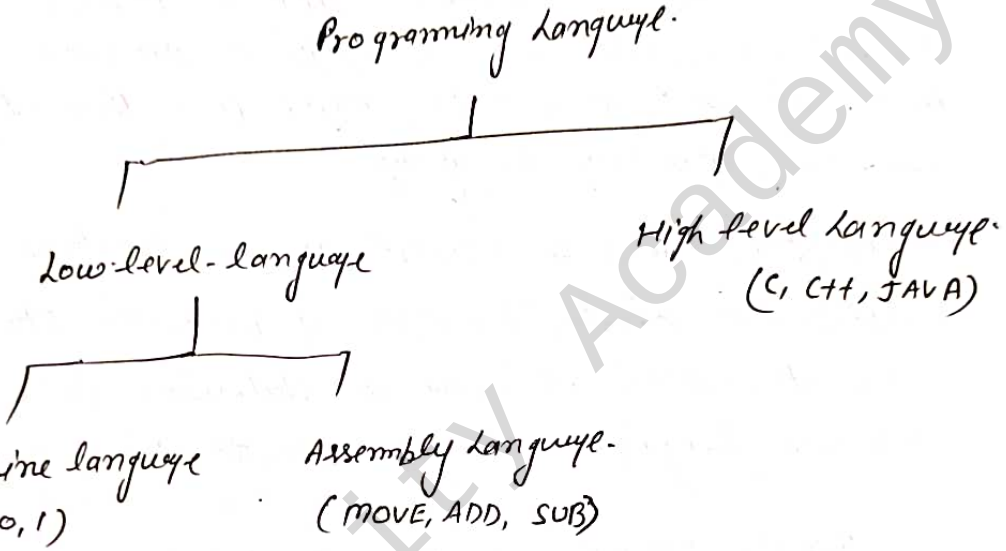
6) Error Detecting: OS Produce error message for any software and hardware failure.

7) Provide the Translator: Provide the assemble to translate each Instruction in binary form (0,1).

1.2.4. Concept of Assembler, Compiler, Interpreter, Loader and linker.

i) Concept of Programming languages.
ii) Concept of translators.
iii) Concept of Loader and Linker.

i) Concept of Programming Language.



a) Machine Language: Machine language is composed of (0,1) binary digit. Machine language is only language a computer is capable to understand.

b) Assembly Language: Assembly Language uses alpha-Numeric code instead of binary digit. It is easy to remember than machine language. Assembly language is machine dependent language.

c) High level Language: It is machine independent lang. it is easy to read, write and maintain as it is written in english like words. HLL is portable.

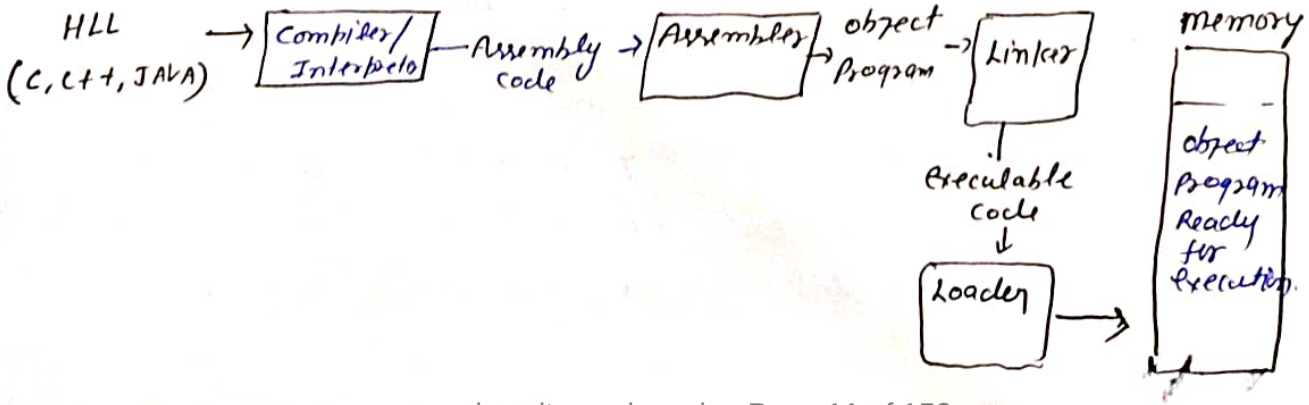
2) Concept of Translator: The translators are just computer program which accept a program written in high level or low-level language and produce equivalent machine language as output. there are three translors:

- Assembler: used to convert Assembly code in to machine language.
- Compiler: used to convert HLL to machine language. Compiler searches all the errors of program and list them. if the program is error free then it converts code into machine language.
- Interpreter: used to convert HLL to machine language. Interpreter checks the error of program statemnt by statemnt. After checking one statement it convert into machine language and the execute that statement.

Note:

Machine language →	No need of translator
Assembly language →	Assembler used to convert Assembly to machine lang.
High level language →	Compiler or ~~Interpreter~~ Interpreter used to convert HLL to low level language.

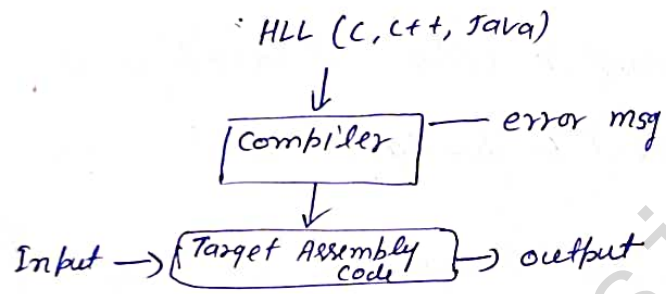
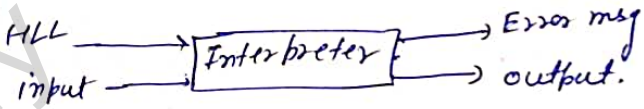
3) Concept of Linker and Loader.



Linker: A linker combine object file generated compiler into a single executable (.exe) file. A linker is also responsible to link and combine all the module of program if written seperately.

Loader: A loader load the program into main memory from storage device. the operating system calls the loader when needed.

Difference between Compiler and Interpreter:

Compiler	Interpreter
	
1- Compiler translate source code into object code as whole.	1. Interpreter translate statments of source code one by one and execute immediatly.
2- It create object file.	2. It does not create object file.
3. Execution is fast	3. Execution is slow.
4- Program not require to ~~run~~ translate each time to run the program.	4- Program require to translate each time to run.
5 Not easy to correct error	5 Eeay to correct mistake in source code.
6. Most HLL uses compiler. eg. C, C++, FORTRAN	6. few languages uses Interpreter eg. lisp, Python, Basic. etc.

1.3. Idea of Algorithm.

The word algorithm comes from the name of a Persian author, Abu Ja'far who wrote a text book on mathmatics Algorithm has come to refer to a method that can be used by computer for the solution of problem.

An algorithm is a finite set of instructions to solve any problem. Every algorithm must follow following criteria:

1. Input : zero or more.
2. Output : atleast one output is produced
3. Definiteness : Each Instuction must be clear and unambiguous.
4. Effectiveness : Each algorithm must be produce effective output as desire.
5. Finiteness : Eeach algorithm must be terminate after finite number of steps.

1.3.1 Representation of Algorithm.

Example:

Write Algorithm to Add two number

1. start
2. input A, B
3. calculate sum = A+B
4. Display sum
5. stop.

Steps to write Algorithm.

1- start.
2. Input
3. Processing / calculation
4- output
5- stop.

Example 2. write An Algorithm to calculate Average of 5 numbers

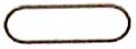
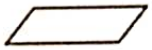
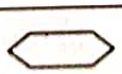
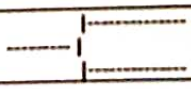
1- Start
2. Input A, B, C, D, E
3 Calculate SUM = A+B+C+D+E
4 Calculate Avg = sum/5.
5- Display Avg.
6. Stop.

Example-3. write an algorithm to find largest among three different numbers.

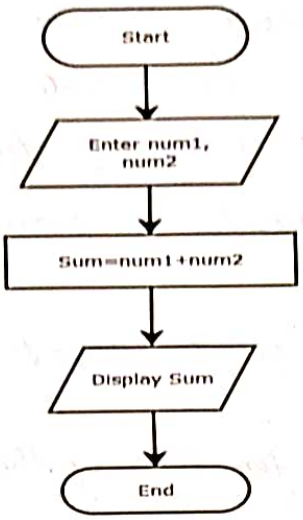
1. Start
2. Input A, B, C
3. if A > B
    if A > C
        Display A is largest
    else.
        Display C is largest
else
    if B > C
        Display B is largest
    else
        Display C is largest.
4- Stop.

1.3.2. Flow chart.

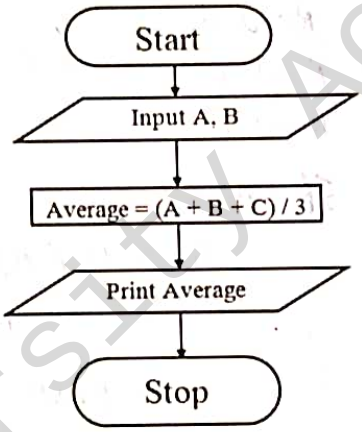
A pictorial representation of an algorithm is called flow chart. In flow chart the steps in the algorithm are represented in the form of different shapes, for example.

Symbol Name	Symbol	function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
		Off Page Connector
		Predefined Process /Function Used to represent a group of statements performing one processing task.
		Preprocessor
		Comments

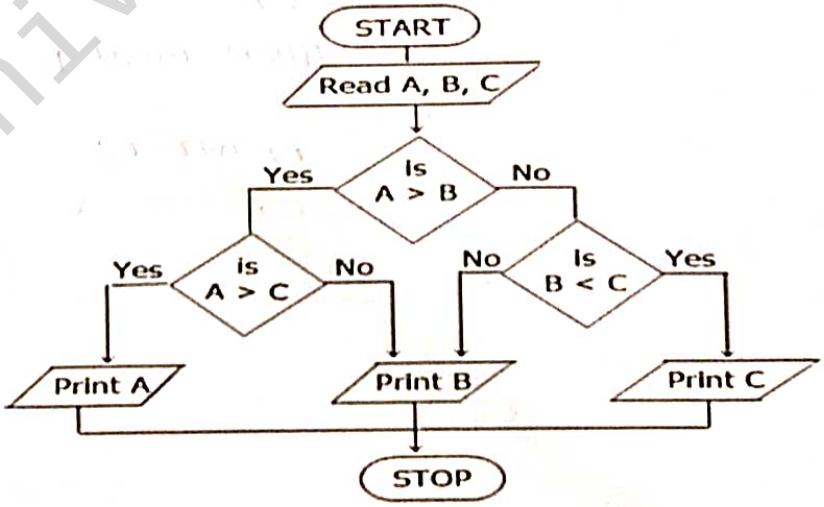
Example 1: Draw the flow chart to add two numbers



Example 2: Draw the flow chart to find average of Three numbers



Example 3: Draw flowchart to find largest of three numbers



1.3.3 Pseudo Code

Pseudo Code is a combination of two words Pseudo and Code. Pseudo means imitation and code refers to Instruction. Pseudo Code is not a real programming code. It is generic way of describing an algorithm without using any specific programming language-related notations.

Pseudo Code is text based detail design tool.

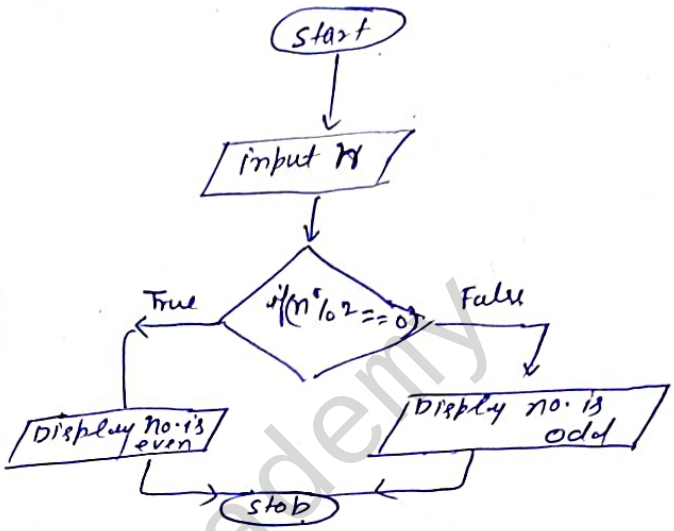
Example: Pseudo code for finding the largest of three number

```
PSEUDOCODE  BiggerOfThree:
            Read A;
            Read B;
            Read C;
            If (A>B)
                THEN IF(A>C)
                        THEN Print A;
                     ELSE  Print C;
                     END IF;
            ELSE IF(B>C)
                     THEN Print B
                 ELSE
                     PRINT C;
                 ENDIF;
            ENDIF;
            END.
```

Exercise

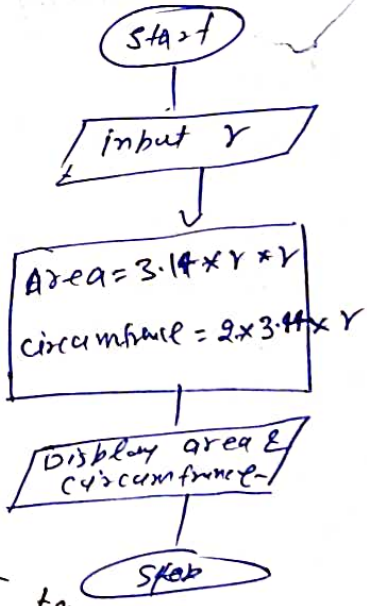
1. Write an algorithm and draw flow chart to check number is even or odd.

1. Start
2. input n
3. if (n%2==0)
   Display number even
   else
   Display number is odd
4. stop



2. write an algorithm and draw flow chart to calculate area and circumferance of circle.

1. start
2. input r
3. Calculate area = 3.14 * r * r
4. Calculate circumference = 2 * 3.14 * r
5. Display area and circumferance.
6. Stop.

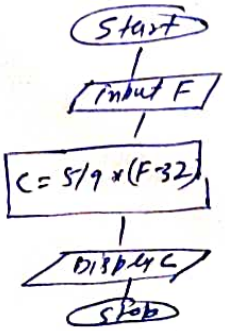


3. write an algo and drow flow chart to convert tem F into C.

$$C = 5/9 * (F-32)$$

Algo

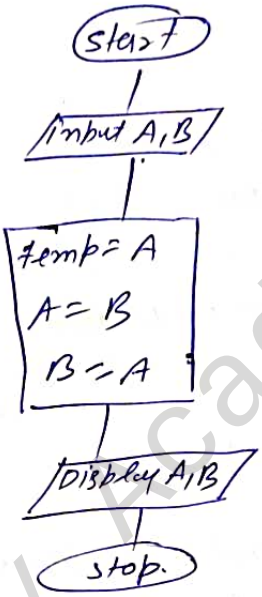
1 start
2 input F
3 C = 5/9 * (F-32)
4 Display C
5 Stop.



4. write an algorithm and draw flow chart to swapping of two number using third variable and without using third variable.

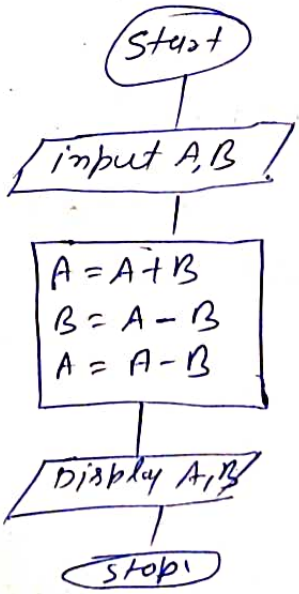
(I) using third variable.

1. Start
2. input A, B
3. temp = A
   A = B
   B = Temp
4. Display A, B.
5. Stop



(II) without using third variable.

1. Start
2. input A, B
3. A = A + B
   B = A - B
   A = A - B
4. Display A, B
5. Stop.



1.4. Programming Basics ('C' Programming)

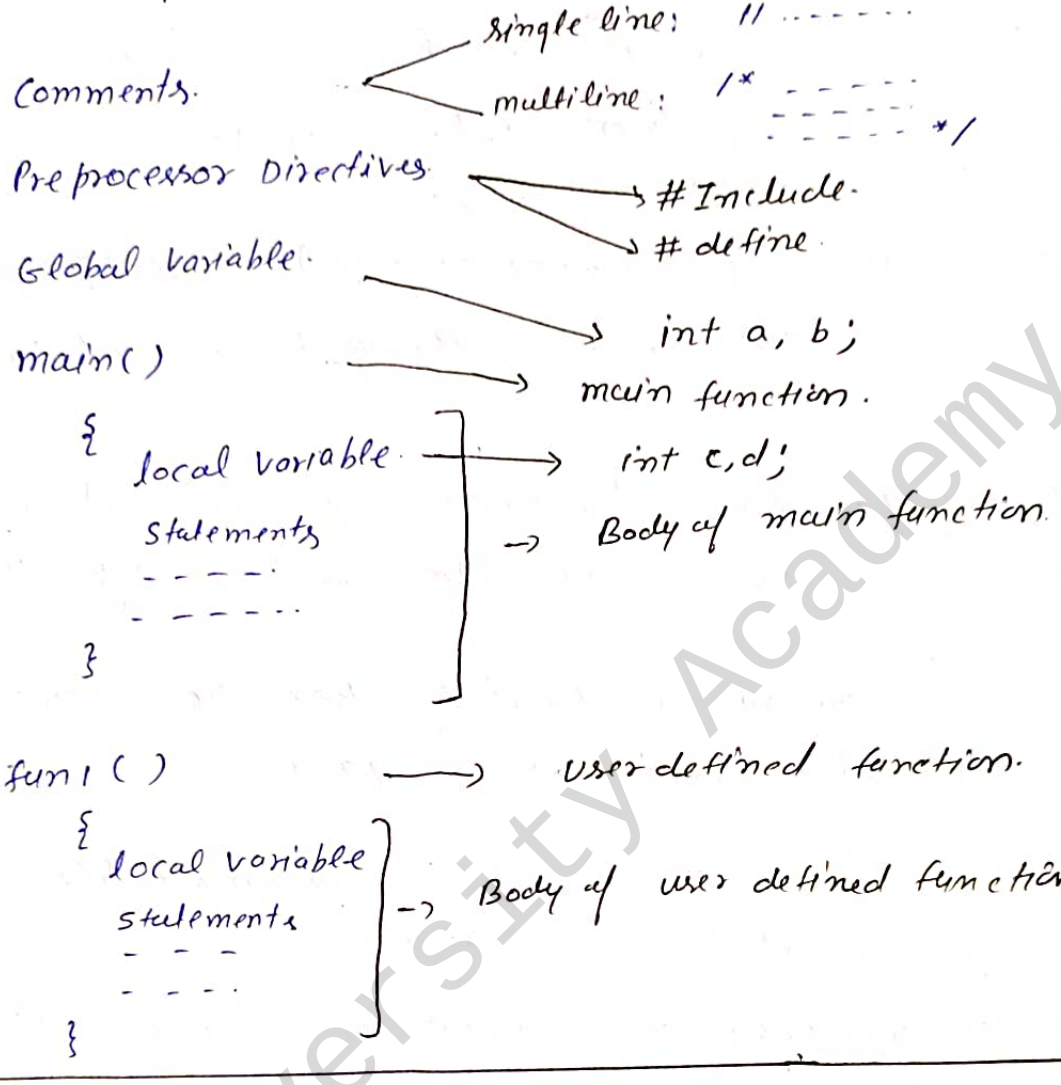
'C' is a general purpose high level language that are originally developed by Dennis Ritchie in 1972. C language is generated in following hierarchy.

1. CPL (Combined programming language) was developed by David Barron in 1963 at University of London and Cambridge University
2. BCPL (Basic Combined Programming language) was developed by Martin Richards at Cambridge University in 1967.
3. B language was developed by Ken thompson at AT & T Bell laboratories in 1969.
4. C language was developed by Dennis Ritchie in 1972 at AT & T Bell laboratories.
5. Finally 1989 the standard for C language was Introduced as ANSI C.

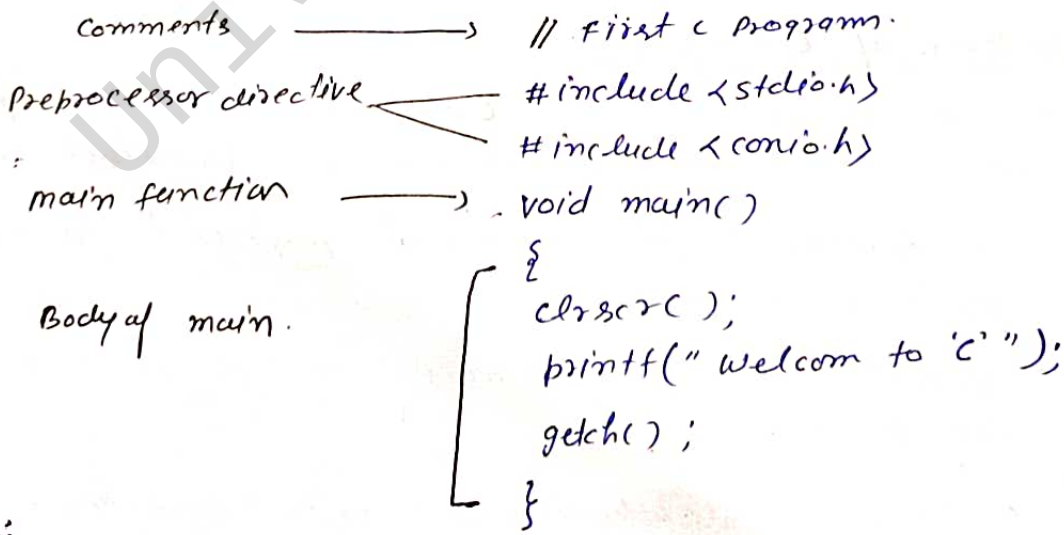
Importance of C Language:

1. C is middle-level Language, 'C' combine the features of both H-L-L and L-L-L (High level language and low level language)
2. C is structured programming-language and highly portable language.
3. C support bit-level programming and direct access to memory using pointer.
4. C language directly interact with Hardware.

1.4.1 Structure of a C program.



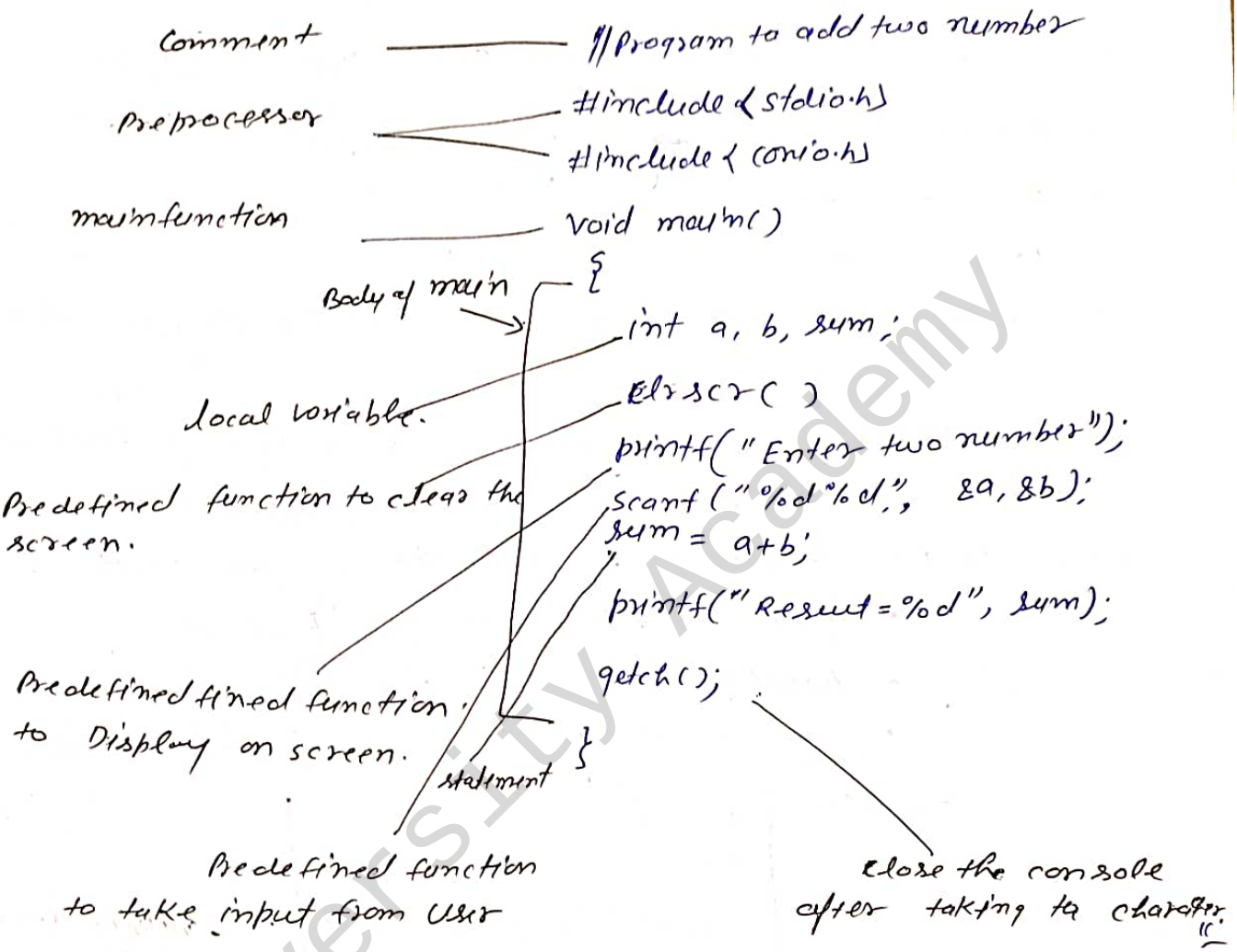
~~functic~~ Example.



clrscr(), getch() are Predefined function defined in conio.h
printf() is Predefined function defined in stdio.h.

1.4.2. Writing and executing the C Program.

Example. Write a program in C to add two number



Example 2. Write a program to ~~get~~ Calculate average of ~~five~~ number

```
#include<stdio.h>
void main()
{
 int a, b, c, d, e;
 float sum, avg;
 printf("Enter the five number");
 scanf("%d%d%d%d%d", &a, &b, &c, &d, &e);
 sum = a+b+c+d+e;
 avg = sum/5;
 printf("Average of numbers = %f", avg);
}
```

Tools for creation and execution.

1. Turbo C
2. GCC
3. Visual studio Code
4. Eclipse
5. Netbeans
6. Code::Blocks
7. Codelite.

1.43 Error in C Program.

In C Programming, we face different kind of errors. These errors can be categorized into different types.

1. Compile time Error → Syntax error
   → Semantic error.
2. Run time error
3. Logical error
4. Linker Error

Syntax error : Syntax error arrise during the compilation of program. These errors are mainly occurred due to mistakes while typing the program. Commonly occured syntax error are

- if we miss the Parenthesis '}'
- forget the declaration of variable.
- missed the (;) at the end of the statement.

Example.

```
#include<stdio.h>
void main()
{
printf("Hello word")
}
```

← missing semicolon. Syntax error.

Semantic error : This kind of error occurs when program is syntatically correct but has no meaning. this is like gramatical mistak in english language.

- Use of uninitialize variable.
- Data type compatibility
- Error in expression eg. a+b=sum;

```
#include<stdio.h>
void main()
{
int a, b, sum;
a=5; b=6;
a+b=sum; // semantic error.
printf("sum=%d", sum);
}
```

Run time error.

This kind of error occurred during the execution of the program. This is not compilation error. the division by zero is common example of run time error.

- divide by zero eg. x = a/0;
- Trying to open file which is not created
- lack of free memory space.

```
#include<stdio.h>
void main()
{
 int a=2, b;
 b= a/0;     // run time error.
 printf("the value of b = %d", b);
}
```

Logical error:

The logical error is an error that leads to an undesired output. These error produce incorrect output but they are error-free known as logical error.

- Semicolon after loop.
- Bracket is not used in formula.
- Incorrect use of operators.

```
#include<stdio.h>
void main()
{
 int i;
 for(i=0; i<=10; i++); // logical error.
   {
     printf("%d", i);
   }
}
```

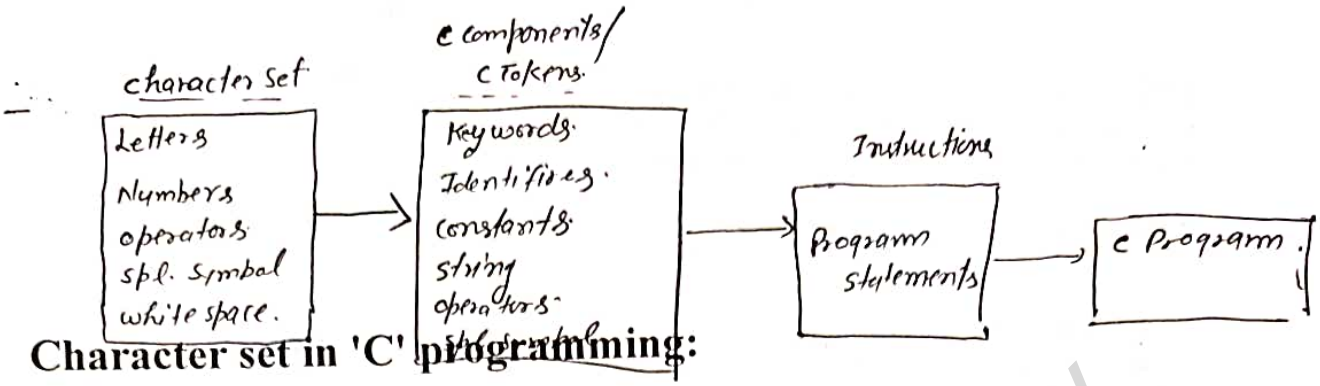
Linker Error

Linker errors are mainly generated when the executable file is not created. this can be happen due to wrong function prototyping or uses of wrong header file.

* if "main" is written as "Main"
* wrong header file.
* wrong function prototyping.

```
#include<stdio.h>
void Main() // linker error.
{
int a = 78;
printf (" value of a = %d ", a);
}
```

1.4.4 Component of C language:



Character set in 'C' programming:

1. Letters
   - Uppercase characters (A-Z)
   - Lowercase characters (a-z)
2. Numbers
   - All the digits from 0 to 9
3. White spaces
   - Blank space
   - New line
   - Carriage return
   - Horizontal tab
4. Special characters and operators.
   - Special characters in 'C' are shown in the given table,

, (comma)	{ (opening curly bracket)
. (period)	} (closing curly bracket)
; (semi-colon)	[(left bracket)
: (colon)	] (right bracket)
? (question mark)	((opening left parenthesis)
' (apostrophe)	) (closing right parenthesis)
" (double quotation mark)	& (ampersand)
! (exclamation mark)	^ (caret)
|(vertical bar)	+ (addition)
/ (forward slash)	- (subtraction)
\ (backward slash)	* (multiplication)
~ (tilde)	/ (division)
_ (underscore)	> (greater than or closing angle bracket)
$ (dollar sign)	< (less than or opening angle bracket)
% (percentage sign)	# (hash sign)

C Tokens

TOKEN is the smallest unit in a 'C' program. It is each and every word and punctuation that you come across in your C program. The compiler breaks a program into the smallest possible units (tokens) and proceeds to the various stages of the compilation.

A token is divided into six different types,

- Keywords
- Identifiers
- Constants
- Strings
- Operators
- Special Characters

Keywords

There are a total of 32 keywords in 'C'. Keywords are written in lowercase letters.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	short	float	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Identifiers

An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc. Identifiers are the user-defined names consisting of 'C' standard character set.

Each identifier must have a unique name. Following rules must be followed for identifiers:

1. The first character must always be an alphabet or an underscore.
2. It should be formed using only letters, numbers, or underscore.
3. A keyword cannot be used as an identifier.
4. It should not contain any whitespace character.
5. The name must be meaningful.

Valid Identifiers

Number
Delhi_24
Count4

Invalid Identifiers

New Delhi
55Delhi
Count#

Constant there are two type of constant

1. Primary Constant ——— (i) integer.
   (ii) Real number
   (iii) character
2. Secondary Constant ——— (i) Array
   (ii) String
   (iii) Pointer
   (iv) structure
   (v) Union.

Here we only focus on primary constant and data type.

Integer constants

An integer constant is nothing but a value consisting of digits or numbers. These values never change during the execution of a program. Integer constants can be octal, decimal and hexadecimal.

- Decimal constant contains digits from 0-9 such as,
  Example, 111, 1234
- Octal constant contains digits from 0-7,
  Example, 012, 065
- Hexadecimal constant contains a digit from 0-9 as well as characters from A-F
  Example, 0X2, 0XBCD

Character constants

A character constant contains only a single character enclosed within a single quote ('). We can also represent character constant by providing ASCII value of it.

Example, 'A', '9'

Real Constants

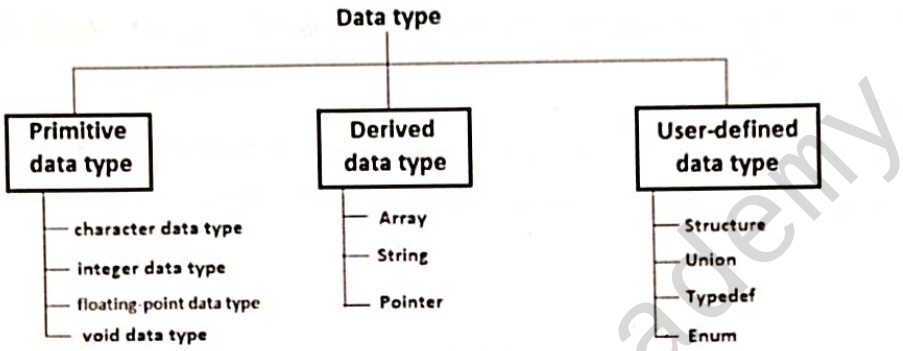
Like integer constants that always contains an integer value. 'C' also provides real constants that contain a decimal point or a fraction value.

Example, 202.15, 300.00

Data types

'C' provides various data types to make it easy for a programmer to select a suitable data type as per the requirements of an application. Following are the three data types:

- Primitive data types
- Derived data types
- User-defined data types



Following table displays the size and range of each data type.

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf

Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.

Note: In upcoming lecture we will discuss about Derived and user define data types

String constants

A string constant contains a sequence of characters enclosed within double quotes ("").

Example, "Hello", "Programming"

1.4.5. **Standard C Input and Output:** Input means to provide the program with some data to be used in the program and Output means to display data on screen or write the data to a printer or a file there are following functions used for input and output.

- scanf() and printf() functions
- getchar() and putchar() Functions
- gets() and puts() Functions

(1) scanf() and printf() function:

the header file stdio.h contain the definition of printf() and scanf() function. the scanf function can be written as :

```
scanf("control string", address variable 1, address variable 2);
```

eg. `scanf("%d %d", &a, &b);`

The printf function written as:

```
printf("control string");
printf("control string", variable 1, variable 2);
```

eg.

```
printf("Welcome to C");
printf("a=%d, b=%d", variable 1, variable 2);
```

Example: Program to check number is even or odd:

```
#include<stdio.h>
void main()
{
int num;
printf("Enter any number");
scanf("%d", &num);
if(num%2==0)
    printf("%d is even number", num);
else    printf("%d is odd number", num);
}.
```

(ii) getchar() and putchar() : (uses header file stdio.h).

getchar() function use to get input a character from keyboard. this reads only single character at a time.

Syntax:

```
getchar();
```

putchar(): putchar() function is used to write a character on screen. this function puts only single character at a time.

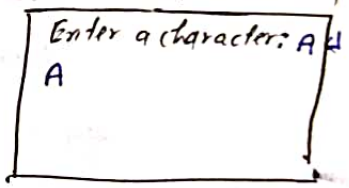
Syntax: putchar(variable);

Note: Above bothe function generally used in ~~function~~ Loop.

Example

```
#include<stdio.h>
void main()
{
  int c;
  printf("Enter a character:");
  c=getchar();
  putchar(c);
}
```

output screen



(iii) gets() and puts() function: (uses header file stdio.h)

gets() function used to get some character followed by ~~enter~~ newline or EOF. All the character stored into string. gets() allows to user to enter the space separated string.

puts() function similar to printf function. puts() function to ~~print~~ ~~puts~~ write the string and newline to screen.

Example

```
#include<stdio.h>
void main()
{
char *str[100]
printf("Enter a value:");
gets(str);
printf("\n you entered:");
puts(str);
}
```

output screen

1.4.6 Variable and Memory Address (location)

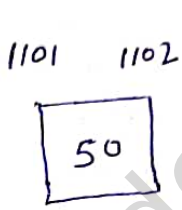
A variable is nothing but a name given to a storage area that our program can manipulate. Each variable has a specific data type which determine the size and layout of variables memory.

Variables Example.

```
int counter = 50;
```

for 16 bit compiler int will occupy 2 byte.
for 32 bit compiler int will occupy 4 byte

Memory location.



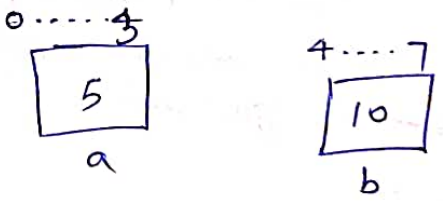
counter ← variable name

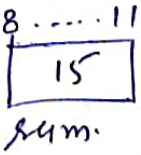
the address of variable can be identify by first byte of variable address.
e.g 1101 is address of variable counter.

Example.

```
#include<stdio.h>
void main()
{int a=5, b=10, sum;
sum = a+b;
printf("sum=%d", sum);
}
```

Assume. int will occupy 4 byte.





address of a is 0
" b is 4
" sum is 8

=

1.4.7 Storage Class:

Storage class in clanguge used determine lifetime, visibility, memory locatin and initial value of variable.

There are four different storage class in C.

(i) Auto (ii) register (iii) static (iv) extern.

(1) auto storage class: The keyword used for defining automatic storage class is "auto", ~~every~~ local and function.

All the variable declared inside a function or ablock is by defaut auto storage class if not anentioned any name of storage class.

example

```
#include<stdio.h>
void main()
{
int a=10, b;
printf(" a=%d, b=%d ", a, b)
}
```

output screen.

a=10
b=2357

(2) register storage class: this storage class used kywork "register" to declare the variable, the compiler tries to store this type of variable in register memory. the default value of register variable is ~~(0)~~ garbage

example

```
#include<stdio.h>
void main()
{
register int a=10, b;
printf("a=%d, b=%d", a, b)
}
```

output screen

a= 10
b=1013

Static storage class: The keyword used for this variable is "static", A static variable tells the compiler to persist (save) the variable until the end of program.

eg.

```
#include<stdio.h>
void test();
void main()
{
  test();
  test();
  test();
}
void test()
{
  static int a=0;
  a=a+1;
  printf("%d\t",a);
}
```

output

1 2 3

if we remove the static keyword result will be.

1 1 1

Extern storage class: the keyword "extern" is used with a variable to inform the compiler that this variable is declared somewhere else

~~Example~~ External storage class is used when we have global function or variable which are shared between two or more file.

Example.

original.c

```
#include<stdio.h>
int i=48
```

main.c

```
#include<stdio.h>
extern int i;
void main()
{
printf("value of i=%d",i);
}
```

output

value of i=48

Properties / Storage Class	Storage	Default Initial Value	Scope	Life
Automatic	Memory	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
Register	CPU Registers	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
Static	Memory	Zero	Local to the block in which the variable is defined	Value of the variable continues to exist between different function calls
External	Memory	Zero	Global	Till the program's execution doesn't come to an end

Important Questions

1. Draw the block diagram of a computer system. Explain its different components with Example
2. Draw the memory hierarchical structure of computer system.
3. What do you mean by operating system?
4. Describe the functionalities of operating system.
5. Differentiate between:
    a. Compiler, Interpreter and assembler
    b. Linker and Loader
6. What are the good characteristics of an algorithm?
7. While compiling a code, write the name of two syntax and two logical errors.
8. Write an algorithm and draw a flowchart to find the sum of digits of an integer number entered by the user.
9. Write an algorithm and draw a flowchart to reverse an integer number entered by the user.
10. Defined data types in C. Discuss primitive data types in terms of memory occupied, format specifier and range
11. Name different storage class with one example of each.
12. List the component of C languages.

Unit II

Arithmetic Expression & Conditional Branching.

2.1 Expressions and Operators:

An operator specifies an operation to be performed on variable or constant that hold the value. An operands is a data item on which an operator acts.

C language includes a large number of operators which as as.

(i) Arithmetic operators
(ii) Assignment operators.
(iii) Increment and decrement operators.
(iv) Relational operators
(v) Logical operators.
(vi) Conditional operators.
(vii) Bitwise operators.
(viii) other operators (a) Comma operators.
(b) sizeof operators.

(i) Arithmetic Operators: An arithmetic operator performs mathmatical operations such as addition, subtraction, multiplication division etc. on numerical value (constant and variable).

operators	meaning.
+	addition or unary plus.
-	subtraction or unary minus
*	multiplication.
/	division.
%	modulo (Remainder)

Example:

```
#include<stdio.h>
void main()
{
int a=10, b=7, S, A, M, D, R;
A=a+b;
S=a-b;
M=A*b;
D=a/b;
R=a%b
printf("%d, %d, %d, %d, %d", A, S, M, D, R);
}
```

output: 17, 3, 70, 1, 3

Note: % (modulo) operator works only with integers.

(ii) Assignment operators: An assignment operator is used for assigning a value to a variable.

operator	Example	same as:
=	a=b	a=b
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b.

```
#include <stdio.h>
void main()
{
int a=7, b=4, c;
c=a;
printf("c=%d", c);
c+=b;
printf("c=%d", c);
c*=b;
printf("c=%d", c);
}
```

output:

```
c=7
c=11
c=44
```

(iii) Increament and decrement Operators: It uses operator (++) or (--), ++ means increment value by one and -- means decrement value by one. this operator used with only variables.

there are two types of increment and decrement operator (1) Pre-fix
(2) Post fix.

Prefix: Here first the value of variable is incremented or decremented by one, then the new value is assign to the variable.

y = ++x

x = x+1
y = x

y = --x

x = x-1
y = x

example:

```
#include<stdio.h>
void main()
{
int x=8;
printf("x = %d \n", x);
printf(" x = %d \n", ++x)
printf("x = %d \n", x);
printf(" x = %d \n", --x);
printf(" x = %d \n", x);
}
```

output

x = 8
x = 9
x = 9
x = 8
x = 8

Postfix: Here first the value of variable is used in operation and then increment or decrement operation performed

y = x++

y = x
x = x+1

y = x--

y = x
x = x-1

Example:

```
#include<stdio.h>
void main()
{
int x=8;
printf("x = %d \n", x);
printf("x = %d \n", x++);
printf("x = %d \n", x);
printf("x = %d \n", x--);
printf("x = %d \n", x);
}
```

output

x = 8
x = 8
x = 9
x = 9
x = 8

IV. Relational operator: A relational operator checks the relationship between two operands. if the relation is true it return '1', if the relation is false it return value '0'. it used in decision making and Loop in C Programming.

operator	Meaning	Example: (Assume a = 5, b = 3)	
==	Equal to	a == b	is evaluated to 0
>	greater than	a > b	is evaluated to 1
<	less than	a < b	is evaluated to 0
!=	Non equal to	a != b	is evaluated to 1
>=	greater than or equal to	a >= b	is evaluated to 1
<=	less than or equal to	a <= b	is evaluated to 0

Example:

```
#include<stdio.h>
void main()
{
int a=5, b=5, c=10;
printf("%d", a==b);
printf("%d", a==c);
printf("%d", a>b);
printf("%d", a>c);
printf("%d", a<b);
printf("%d", a<c);
printf("%d", a!=c);
printf("%d", a>=b);
printf("%d", a<=c);
}
```

output

```
1
0
0
0
0
1
1
1
1
```

(V) Logical Operator: An expression containing logical operators return either 0 or 1 depending whether expression result true or false. this operation commonly used in decision making programming.

operator	meaning.
&&	logical AND true only if all operands are true.
||	logical OR true if either operand are true.
!	logical NOT true only if operand is '0'.

Example.

```
#include<stdio.h>
void main()
{
int a=5, b=5, c=10, result;
result = (a==b) && (c>b);
printf("%d", result);
result = (a!=b) || (c<b);
printf("%d", result);
reset = !(a==b);
printf("%d", reset);
}
```

output.

1

0

0

(VI) Conditional operator: conditional operator is ternary operator denoted by (? and :) which requires three expression or operand and this is written as.

Test exp ? expression1 : expression2.

eg: a>b ? a : b

Example-

```
#include<stdio.h>
void main()
{
int a = 12, b = 7, max;
max = a>b ? a : b;
printf("max = %d", max);
}
```

output

max = 12

(VII) Bitwise Operators: The Bitwise operator are used for operation on individual bits it operate on integer only.

In turbo C (16-bit System)
int size is 2byte = 16bit
short int size 1byte = 8bit

Bitwise operator	meaning.
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise complement
<<	shift left.
>>	shift Right.

Bitwise AND: output of Bitwise AND is 1 if the corresponding bits of two operands is 1. if either bit of an operand is 0 the result of corresponding bit is evaluted to 0.

e: 12 = 00001100
25 = 00011001

bitwise AND of 12 and 25 are.

```
  00001100  (12)
& 00011001  (25)
------------
  00001000  (8)
```

```
#include<stdio.h>
void main()
short int a=12, b=25
    printf("output = %d", a&b);
}
```

output

output = 8

Bitwise OR : output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. otherwise result is 0.

ex:

```
  00001100 (12)
| 00011001 (25)
----------------
  00011101 (29)
```

Example:

```
#include<stdio.h>
void main()
{
int a=12, b=25;
printf("output = %d", a|b)
}
```

output

output = 29

Bitwise XOR The result of XOR operator is 1 if the corresponding bits of two operands are opposit.

```
  00001100 (12)
^ 00011001 (25)
----------------
  00010101 (21)
```

Bitwise Complement : it change 0 to 1, and 1 to 0.

bitwise complement of 35

binary of 35 into 8-bit formate.

35 = | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

Most significant Bit — least significant bit.

MSB: 0 → positive. 1 → Negative.

LSB: 0 → even 1 → odd

one's complement ⇓

128	64	32	16	8	4	2	1
1	1	0	1	1	1	0	0

MSB — LSB

$-128+64+0+16+8+4+0$

$-64+16+8+4+0 =$

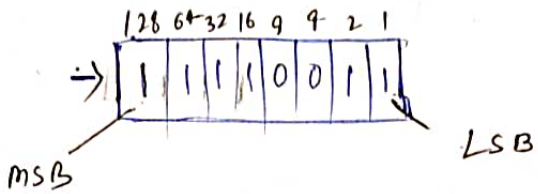
$-64+28 = -36$

one's complement of (-12)

→ 12 = 0 0 0 0 1 1 0 0

-12 = → 0 0 0 0 1 1 0 0

∧ one's complement



~~128 = 64 = 32 = 16 = 2 = 1~~

two's complement

~~0 0~~

1 1 1 1 0 0 1 1

\+ 1

1 1 1 1 0 1 0 0 binary of (-12)

↓ ~(-12)

0 0 0 0 1 0 1 0 = 11 ✓

Note. Bitwise complement of N = ~N (Represented in 2's complement form)

2's complement of ~N = -(~N)+1) = -(N+1)

~(35) = -(35+1) = -36

~(-12) = -(-12+1) = -(-11) = 11

Shift operators

1. Left shift (<<)

212 = 1 1 0 1 0 1 0 0 (binary)

212 << 1 = 1 1 0 1 0 1 0 0 0

212 << 4 = 1 1 0 1 0 1 0 0 0 0 0 0

2. Right shift. (>>)

212 = 1 1 0 1 0 1 0 0

212 >> 1 = 0 1 1 0 1 0 1 0

212 >> 2 = 0 0 1 1 0 1 0 1

212 >> 7 = 0 0 0 0 0 0 0 1

Other operator:

i. Comma operator: The comma operator are used to link related expression togathers. eg:

```
int a, b, c=5;
```

(iii) Sizeof operator: the sizeof is a unary operator that return the size of data (constant, variable, array, structure etc)

Example

```
#include<stdio.h>
void main()
{
  int a;
  float b;
  double c;
  char d;
  printf("Sizeof int = %d bytes\n", sizeof(a));
  "  "  "  float = "  "  , sizeof(b));
  "  "  "  double = "  "  , sizeof(c));
  printf("sizeof char = %d bytes\n", sizeof(d));
}
```

Output:

```
sizeof int = 4 bytes
sizeof float = 4 bytes
sizeof double = 8 bytes
sizeof char = 1 byte.
```

2.2. Type Conversion in C

The type cast is basically a conversion from one type to another. there are two type of conversion.

1. Implicit Type Conversion.
2. Explicit Type Conversion.

1- Implicite type conversion:

The type conversion that is done automatically by compiler is known as implicit type conversion. when the expression has more than one data type in ~~Generally~~ such as type promotion take place to avoid loss of data.

All the data type of variable are upgraded to the data type of variable with largest data type.

bool → char → short int → int → long int → float → double → long double

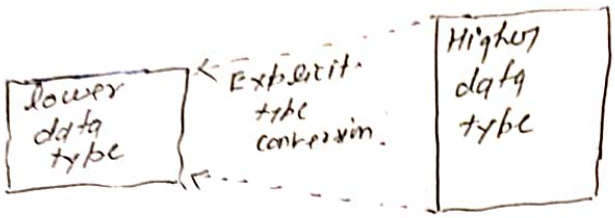
Example:

```
#include<stdio.h>
void main()
{
int x=10;
char y='a';
x=x+y;
float z=x+1.0;
printf("x=%d, z=%f", x, z)
}
```

output:

x=107, z=108.000000

2. Explicit Type Conversion :→ when the user manually changes data from one type to another, this is known as explicit conversion. this type of conversion is also known as type casting.



Syntax: type (expression)

Example:

```
#include<stdio.h>
void main()
{
  double x = 1.2;
  int sum = int(x) + 1;
  printf("sum = %d", sum)
}
```

output: sum = 2

23 Operator Precedence and Associativity

Operator	Description	Precedence level	Associativity
( ) [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Dot operator (Member selection via object name) Arrow operator(Member selection via pointer) Postfix increment/decrement	1	Left to Right
+ - ++ -- ! ~ * & (datatype) sizeof	Unary plus Unary minus Prefix increment/decrement Logical NOT One's complement Indirection Address (of operand) Type cast Determine size in bytes on this implementation	2	Right to Left
* / %	Multiplication Division Modulus	3	Left to Right
+ -	Addition Subtraction	4	Left to Right
<< >>	Left shift Right shift	5	Left to Right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	Left to Right
== !=	Equal to Not equal to	7	Left to Right
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
|	Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
||	Logical OR	12	Left to Right
?:	Conditional operator	13	Right to Left
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	14	Right to Left
,	Comma operator	15	Left to Right

Exercise

Assume stdio.h is included in all programs.

(1)
```
main( )
{
    int a=-3;
    a=-a-a+!a;
    printf("%d\n",a);
}
```
Ans: 6

(2)
```
main( )
{
    int a=2,b=1,c,d;
    c=a<b;
    d=(a>b)&&(c<b);
    printf("c = %d, d = %d\n",c,d);
}
```
Ans: c = 0, d = 1

(3)
```
main( )
{
    int a=9,b=15,c=16,d=12,e,f;
    e=!(a<b||b<c);
    f=(a>b) ? a-b:b-a;
    printf("e = %d, f = %d\n",e,f );
}
```
Ans: e = 0, f = 6

(4)
```
main( )
{
    int a=5;
    a=6;
    a=a+5*a;
    printf("a=%d\n",a)
}
```
Ans: 36

```
(5) main()
    {
        int a=5,b=5;
        printf("%d, %d\t",++a,b--);
        printf("%d, %d\t",a,b);
        printf("%d, %d\t",++a,b++);
        printf("%d, %d\n",a,b);
    }
```

Ans: 6, 5 6, 4 7, 4 7, 5

```
(6) main()
    {
        int x,y,z;
        x=8++;
        y=++x++;
        z=(x+y)--;
        printf("x = %d, y = %d, z = %d\n",x,y);
    }
```

Ans: error

```
(7) main()
    {
        int a=4,b=8,c=3,d=9,z;
        z=a++ + ++b * c-- - --d;
        printf("a = %d,b = %d,c = %d,d = %d,z = %d\n",a,b,c
    }
```

Ans: a=5, b=9, c=2, d=8, z=23

```
(8) main()
    {
        int a=14,b,c;
        a=a%5;
        b=a/3;
        c=a/5%3;
        printf("a = %d, b = %d, c = %d\n",a,b,c);
    }
```

Ans: a=4, b=1, c=0

```
(9) main()
    {
        int a=15,b=13,c=16,x,y;
        x=a-3%2+c*2/4%2+b/4;
        y=a=b+5-b+9/3;
        printf("x = %d, y = %d\n",x,y);
    }
```

Ans: x=17, y=8

```
(10)main()
    {
        int x,y,z,k=10;
        k+=(x=5,y=x+2,z=x+y);
        printf("x = %d,y = %d,z = %d,k = %d\n",x,y,z,k);
    }
```

Ans: x=5, y=7, z=12, k=22

(11)
```
main()
{
    int a;
    float b;
    b=15/2;
    printf("%f\t",b);
    b=(float)15/2+(15/2);
    printf("%f\n",b);
}
```
Ans: 7.000000 14.500000

(12)
```
main()
{
    int a=9;
    char ch='A';
    a=a+ch+24;
    printf("%d,%c\t%d,%c\n",ch,ch,a,a);
}
```
Ans: 65, A 98, b

(13)
```
main()
{
    int a,b,c,d;
    a=b=c=d=4;
    a*=b+1;
    c+=d*=3;
    printf("a = %d,c = %d\n",a, c);
}
```
Ans: a=20, c=16

(14)
```
main()
{
    int a=5,b=10,temp;
    temp=a,a=b,b=temp;
    printf("a = %d,b = %d\n",a,b);
}
```
Ans: a=10, b=5

(15)
```
main()
{
    int a=10,b=3,max;
    a>b?max=a:max=b;
    printf("%d",max);
}
```
Ans: error

(16)
```
#include<stdio.h>
main()
{
    int a=5,b=6;
    printf("%d\t",a=b);
    printf("%d\t",a==b);
    printf("%d  %d\n",a,b);
}
```
Ans: 6 1 6 6

2.4 Conditional Branching: programming are used to make decisions based on the conditions. Conditional statements execute sequentially when there is no condition around the statements. Statement

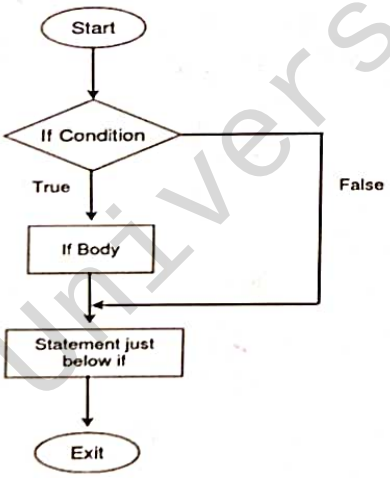
i. if statements
ii. if..else statements
iii. nested if-else statements
iv. if-else-if ladder
v. switch statements

i. **if statement:** it is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
// Statements to execute if
// condition is true
}
```

Flow Chart



Program:

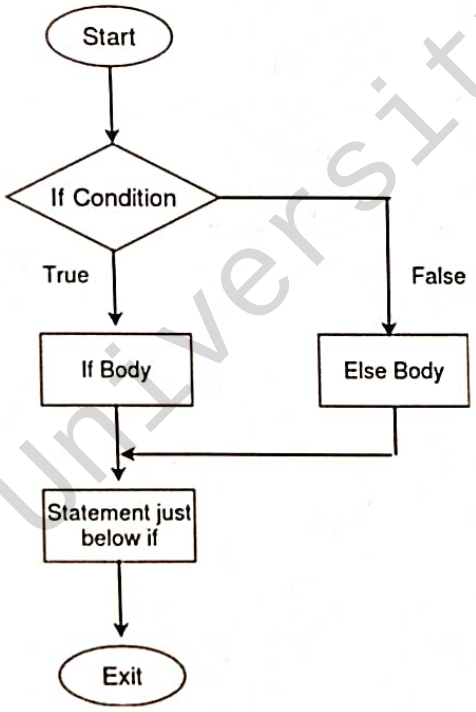
```
// C program to illustrate If statement
#include <stdio.h>
int main()
 {
        int i = 10;
        if (i > 15)
        {
        printf("10 is less than 15");
        }
}
```

ii. **if..else statements :** The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the C else statement

Syntax:

```
if (condition)
 {
 // Executes this block if
 // condition is true
 }
else
 {
 // Executes this block if
 // condition is false
 }
```

Flow Chart



Program:

```
#include <stdio.h>
void main()
{
   int num;
   printf("Enter an integer: ");
   scanf("%d", &num);
   if(num % 2 == 0)
        {
       printf("%d is even.", num);
        }
  else
        {
       printf("%d is odd.", num);
        }
 }
```

iii. **Nested if-else statements:** it means an if statement inside another if statement. i.e, we can place an if statement inside another if statement.

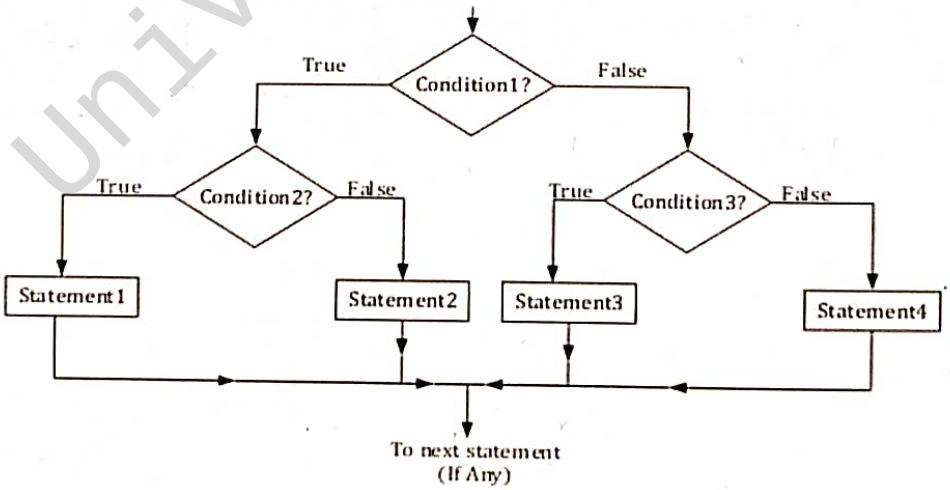
Syntax

```
if (Condition1)
{
    if(Condition2)
    {
        Statement1;
    }
    else
    {
        Statement2;
    }
}
else
{
    if(Condition3)
    {
        Statement3;
    }
    else
    {
        Statement4;
    }
}
```

Program:

```
#include<stdio.h>
void main()
{
        int num1, num2, num3;
        printf("Enter three numbers:\n");
        scanf("%d%d%d",&num1, &num2, &num3);
        if(num1>num2)
        {
                if(num1>num3)
                {
                        printf("Largest = %d", num1);
                }
                else
                {
                        printf("Largest = %d", num3);
                }
        }
        else
        {
                if(num2>num3)
                {
                        printf("Largest = %d", num2);
                }
                else
                {
                        printf("Largest = %d", num3);
                }
        }
}
```

Flow Chart

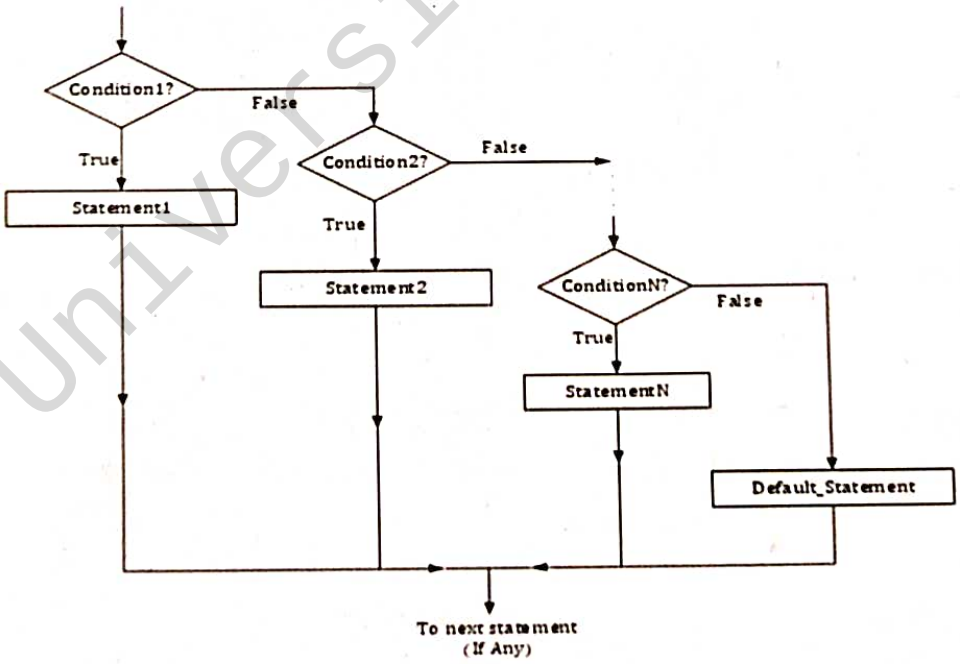


iv. **if-else-if ladder** : if-else-if statement is also known as if-else-if ladder. It is used when there are more than two possible action based on different conditions.

Syntax:

```
if (Condition1)
        {
        Statement1;
        }
else if(Condition2)
        {
                Statement2;
        }
                .
                .
                .
                else if(ConditionN)
                        {
                           StatementN;
                        }
                        else
                        {
                         Default_Statement;
                        }
```

Flow Chart



Program:

C program to print weekday based on given number.

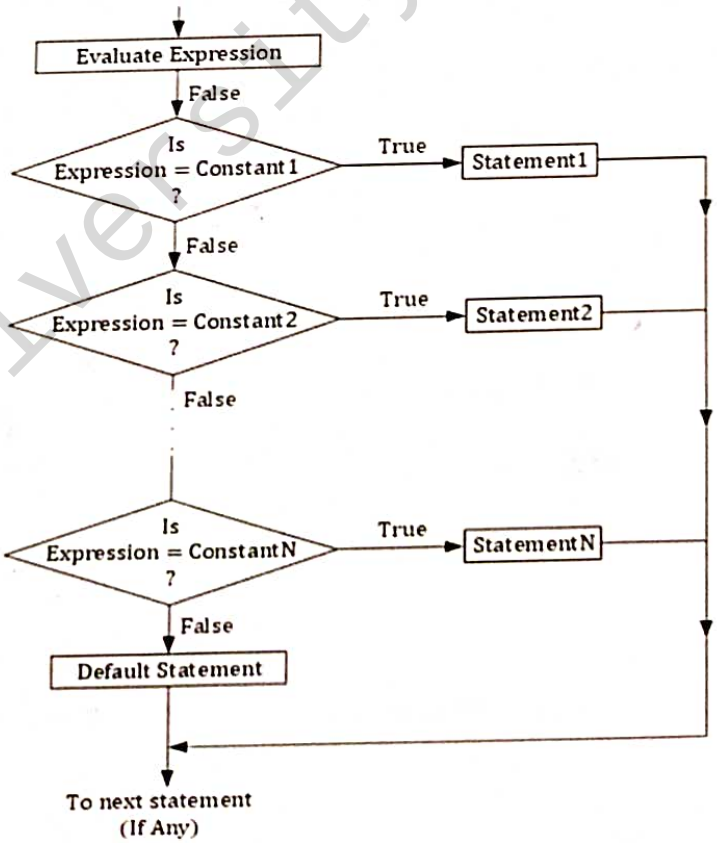
```
#include<stdio.h>
void main()
{
        int day;
        printf("Enter day number: ");
        scanf("%d", &day);
        if(day==1)
        {
                printf("SUNDAY.");
        }
        else if(day==2)
        {
                printf("MONDAY.");
        }
        else if(day==3)
        {
                printf("TUESDAY.");
        }
        else if(day==4)
        {
                printf("WEDNESDAY.");
        }
        else if(day==5)
        {
                printf("THURSDAY.");
        }
        else if(day==6)
        {
                printf("FRIDAY.");
        }
        else if(day==7)
        {
                printf("SATURDAY.");
        }
        else
        {
                printf("INVALID DAY.");
        }
}
```

v. **Switch statement:** A switch case statement is a multi-branched statement which compares the value of a variable to the values specified in the cases. The switch statement allows us to execute one code block among many alternatives. You can do the same thing with the if...else..if ladder.

Syntax:

```
switch (Expression)
{
   case Constant1: Statement1;
                   break;

   case Constant2: Statement2;
                   break;
   .
   .
   .
   case ConstantN: StatementN;
                   break;

   default:  Default Statement;
}
```

Flow Chart:



Program:
C program to print weekday based on given number using switch

```
#include<stdio.h>
void main()
{
        int day;
        printf("Enter day number: ");
        scanf("%d", &day);
        switch(day)
        {
                case 1: printf("SUNDAY.");
                        break;
                case 2: printf("MONDAY.");
                        break;
                case 3: printf("TUESDAY.");
                        break;
                case 4: printf("WEDNESDAY.");
                        break;
                case 5: printf("THURSDAY.");
                        break;
                case 6: printf("FRIDAY.");
                        break;
                case 7: printf("SATURDAY.");
                        break;
                default: printf("INVALID DAY.");
                        break;
        }
}
```

C program to implement simple calculator using switch statement

```
#include<stdio.h>
void  main()
{
        float a,b, r;
        char op;
        printf("Available Operations.\n");
        printf("+ for Addition.\n");
        printf("- for Subtraction.\n");
        printf("* for Multiplication.\n");
        printf("/ for Division.\n");
        printf("Which Operation?\n");
        scanf("%c", &op);
```

```
switch(op)
{
        case '+': printf("Enter two numbers:\n");
                  scanf("%f%f",&a, &b);
                  r = a+b;
                  printf("%f + %f = %f", a, b, r);
                  break;
        case '-': printf("Enter two numbers:\n");
                  scanf("%f%f",&a, &b);
                  r = a-b;
                  printf("%f - %f = %f", a, b, r);
                  break;
        case '*': printf("Enter two numbers:\n");
                  scanf("%f%f",&a, &b);
                  r = a*b;
                  printf("%f * %f = %f", a, b, r);
                  break;
        case '/': printf("Enter two numbers:\n");
                  scanf("%f%f",&a, &b);
                  if(b!=0)
                  {
                        r = a/b;
                        printf("%f/%f = %f",a,b,r);
                  }
                  else
                  {
                              printf("Division not possible.");
                  }
                  break;
        default:  printf("Invalid Operation.");
                  break;
        }
    return(0);
}
```

Important Questions

1. Differentiate assignment and Equality operators in C.
2. Differentiate operator precedence and associativity.
3. Write a program in C to elaborate the use of logical AND logical OR Operator in C.
4. Define the term mixed operands in an arithmetic expression with few examples.
5. Write a program in C to elaborate the use of type casting
6. Explain various types of arithmetic operators in C language with help of example.
7. When precedence of two operators in an arithmetic expression is same, how associativity helps in identifying which operator will be evaluated first. Illustrate it with the example
8. Differentiate between implicit & Explicit type conversion.
9. What are operators? Mention different types of operators in C.
10. Correlate else if ladder switch case statement
11. Write a program that takes two operands and one operator from the user and perform the operation and prints the result by using switch statement.
12. A certain grade of steel is graded according to the following conditions:
    i. Hardness must be greater than 50
    ii. Carbon content must be less than 0.7.
    iii. Tensile strength must be less than 5600

    The grades are as follows:

    Grade is 10 if all the three conditions are met.

    Grade is 9 if condition (i) and (ii) are met

    Grade is 8 if condition (ii) and (iii) are met

    Grade is 7 if condition (i) and (iii) are met

    Grade is 6 if only one condition is met.

    Grade is 5 if none of the conditions are met.

    Write a program, which will require the user to give values of hardness, carbon content and tensile strength of the steel under consideration and output the grade of the steel.

UNIT- III

Loops & Functions

3-1 Iteration and Loops

Iteration is a process where a set of instructions or statements is executed repeatedly for a specific number of time or until a condition met.

Iteratation statements are most commonly known as Loop. there are three types of Looping statement.

(i) While Loop
(ii) do-while Loop
(iii) for Loop

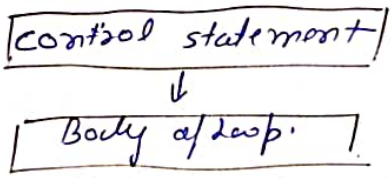
} Loop consit three part. Initialization, condition, Increment/decrement or update.

A Loop consist two part one is "body of Loop" and second is "contral statement". Body of Loop executed until spefified condition becomes false.

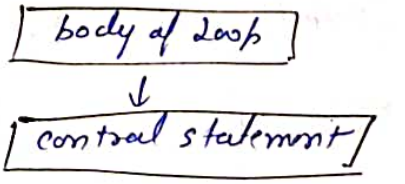
Classification of Loop Depeding upon position of contral statemet in a program a Loop is classified into two types.

(i) Entry Contralled Loop
(ii) Exit Contralled Loop.

In Entry Contralled Loop condition checked before the execution of body of Loop. eg: while, for.



In Exit contralled Loop condition is checked after the execution of body of Loop. eg. do-while.



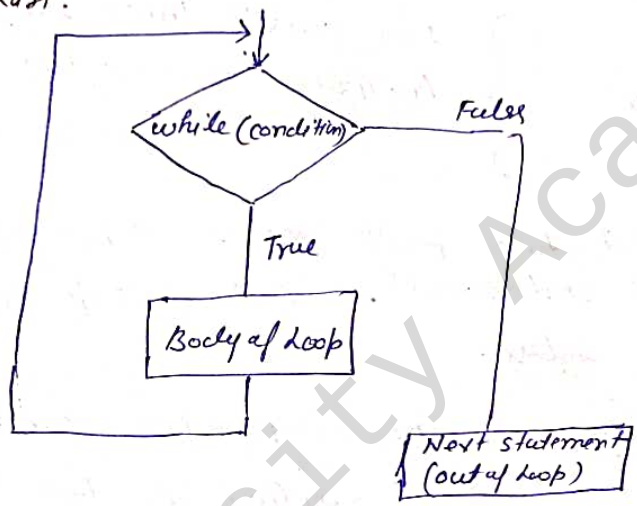
3.1.1 while Loop:

It is a entry controlled loop. In while loop condition is evaluated before the execution of body of loop

Syntax:

```
while (condition)
{
 statement1;
 statement2;
 .
 .
 statement n;
}
```

Flow chart.



Example:1

WAP in C to Print number from 1 to 10.

```
#include<stdio.h>
void main()
{
int i=1;
while (i<=10)
{
 printf("%d \n", i);
 i=i+1;
}
}
```

output:

```
1
2
3
4
5
6
7
8
9
10
```

Example:2:

WAP in C to Print sum of digit of any number.

```
#include<stdio.h>
void main()
{
int n, sum=0, rem;
printf("Enter the number");
scanf("%d", &n);
while (n>0)
{
 rem=n%10;
 sum=sum+rem;
 n=n/10;
}
printf("sum of digit= %d", sum);
}
```

output

```
Enter the number
512 ↵
sum of digit= 8
```

Example.3 Program to find the factorial of any number.

```
#include<stdio.h>
void main()
{
    int n, num;
    long fact=1;
    printf(" Enter the number");
    scanf ("%d", &n);
    num=n;
    if (n<0)
        printf(" no negative number for factorial");
    else
    {
        while (n>1)
        {
            fact = fact * n;
            n--;
        }
        printf(" factorial of given number is =%ld \n", fact);
    }
}
```

output

```
Enter the number 4↵
factorial of given number
= 24
```

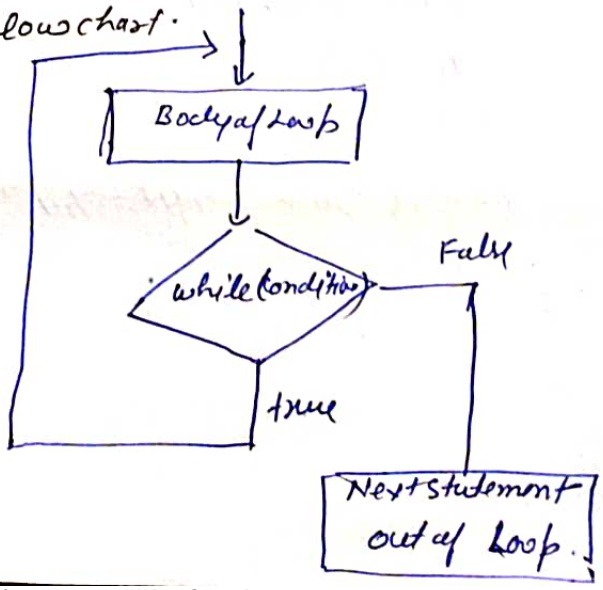
3.1.2 do-while Loop:

A do-while loop is similar to while loop except the condition is always executed after the body of loop.

Syntax:

```
do
{
 statement 1;
  :
 statement n;
} while(condition);
```

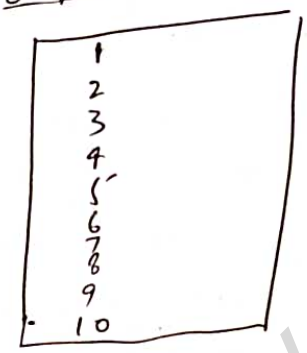
Flowchart:



Example 1: Program to print number from 1 to 10.

```
#include<stdio.h>
void main()
{
    int i=1;
    do
    {
      printf("%d\n", i);
      i++;
    } while(i<=10);
}
```

output



Example 2. Program to print sum of digit of any number. using do-while loop

```
#include<stdio.h>
void main()
{
 int n, sum=0, rem
 printf("Enter the number");
 scanf("%d", &n);
 do
 {
   rem = n % 10;
   sum = sum + rem;
   n = n / 10;
 } while(n>0);
 printf("sum of digit = %d", sum);
}.
```

Example 3: Program in c to print the reverse of any number.

```
#include<stdio.h>
void main()
{
  int n, rev=0, rem;
  printf("Enter a number");
  scanf("%d", &n);
  do
  {
    rem = n%10;
    rev = rev*10 + rem;
    n = n/10;
  } while (n>10);
  printf("Reverse number = %d", rev);
}
```

output

```
Enter a number :
512 ↵
Reverse number = 215
```

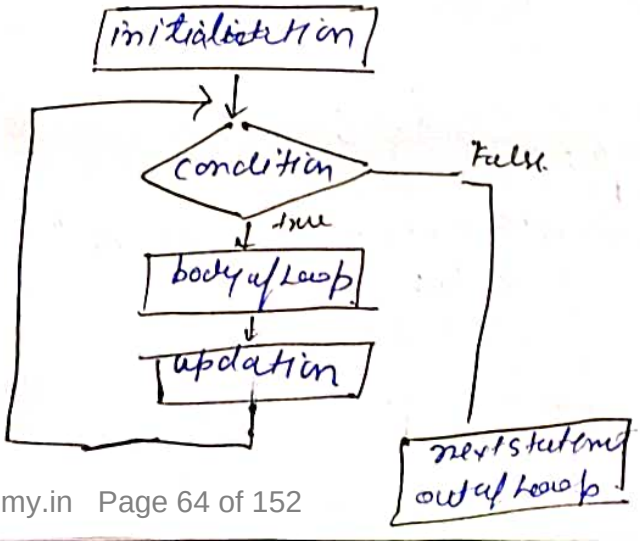
3.1.3 FOR LOOP

'for' loop is entry controlled loop.

it has three expression two semicolons are used to separating these expression.

Syntax:

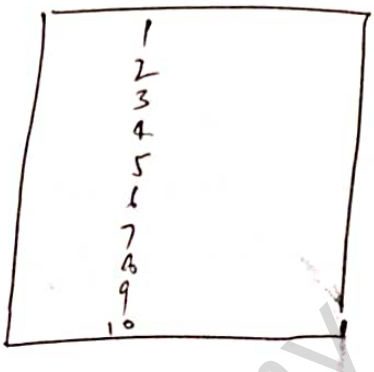
```
for(initialization; condition; updation)
{
  statement 1;
  :
  statement n;
}
```



Example 1: print number from 1 to 10 using for loop

```
#include<stdio.h>
void main()
{
 int i;
for(i=1; i<=10; i++)
  {
     printf("%d\n", i);
  }
```

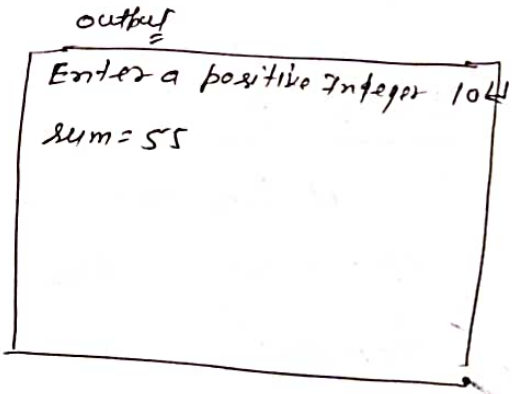
output



Example-2: Program to calculate sum of first n natural number.

```
#include<stdio.h>
void main()
 {
  int num, count, sum=0;
  printf("Enter a positive integer");
  scanf("%d", &num);
   for(count=1; count<=num; count++)
    {
      sum = sum+count;
    }
    printf("sum=%d", sum);
 }
```

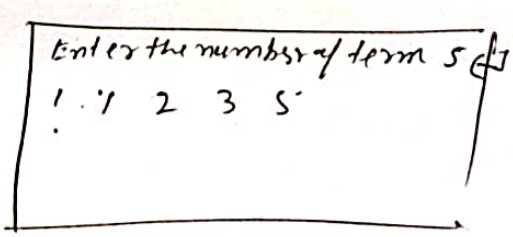
output



Example-3: Program to generate fibbonacci series.
eg: 1, 1, 2, 3, 5, 8, 13, 21 -----

```
#include<stdio.h>
void main()
{
int x, y, z, i, n;
x=0;
y=1;
printf("Enter the number of terms.");
scanf("%d", &n);
for(i=1; i<n; i++)
{   z=x+y;
    printf("%d  ", z);
    x=y;
    y=z
}
printf("\n"); }
```

output



Examples 4. Program to check given number is prime or not.

```
#include<stdio.h>
void main()
{
int n, i, flag=1;
printf("Enter the number");
scanf("%d", &n);
for(i=2; i<n/2; i++)
  {
    if (n%i==0)
      {
        flag=0
        break;
      }
    if (flag==1)
       printf("number prime");
    else
       printf("number is not prime)
  }
}
```

output:

Enter the number 29 ↵
- number is ~~not~~ prime.

3.1.4 Nesting of loops:

When a loop is written inside the body of another loop, then it is known as nesting of loop. Any type of loop can be nested inside any other type of loop.

Syntax

```
for ( ; ; )
 {
    while ( )
     {
       statement 1;
       :
       statement n;
     }
   statements
 }
```

Example: Program to print armstrong number from 100 to 999.

```
#include<stdio.h>
void main()
 {
 int num, n, rem, sum;
 printf("Armstrong number are:");
 for(num=100; num<=999; num++)
  {
     n=num;
     sum=0;
     while(n>0)
       {
         rem= n%10;
         n = n/10;
         sum = sum + (rem*rem*rem);
       }
     if(num==sum)
        printf("%d\n", num)
  }
}
```

```
Armstrong numbers are
153
370
371
407
```

Example 2: C Program to print all prime number bet^n 1 to n

```
#include<stdio.h>
void main()
{
int i, J, n, flag
printf("Enter number");
scanf("%d", &n);
for( i=2; i<=n ; i++)
 {
    flag=1;
    for(J=2 ; J<=i/2; J++)
     {
        if (i % J == 0)
           {
              flag=0;
              break;
           }
     }
   if (flag == 1)
     {
       printf("%d", i);
     }
}
}
```

```
Enter number
2, 3, 5, 7, 11, 13, 17, 19, 23,
31, 37, 41, 43, 47.
```

3.1.5 Break Statement:

Break statement is used inside loops and switch statement when break statement encounter the body of Loop and switch has been terminated and control transfer the outside of body.

Syntax

```
break;
```

the break statement is almost used with "if..else" inside the loop.

Example 1: Break with Loop.

```
#include<stdio.h>
void main()
{
int i;
for(i=0; i<=10; i++)
{
   printf("%d", i);
   if(i==5)
      break;
}
printf("outside of Loop");
}
```

output

```
0 1 2 3 4 5 outside of loop
```

Example 2: Break with switch-case.

```
#include<stdio.h>
void main()
{
int choice;
printf("Enter your choice:");
scanf("%d", &choice);
switch(choice)
{
  case 1: printf("First\n");
          break;
  case 2: printf("second\n");
          break;
  case 3: printf("third\n");
          break;
  default:
          printf("wrong choice");
}
}
```

```
Enter your choice: 2
Second
```

3.1.6. Continue Statement:

The continue statement is used when we want to go to next iteration of loop after skipping some statement of the loop.

Syntax:

```
continue;
```

```
#include<stdio.h>
void main()
{
int i;
for(i=1; i<=10; i++)
  {  if(i==5)
       continue;
     printf("%d\n", i);
  }
```

output

```
1
2
3
4
6
7
8
9
10
```

3.1.7. goto Statement:

This is an unconditional control statement that transfers the flow of control to another part of the program.

Syntax:

```
goto label;
---
---
label:
    statement
    -----
    -----
```

example: Program to print whether the number is even or odd.

```
#include<stdio.h>
void main()
{ int n;
printf("Enter the number");
scanf("%d", &n);
if(n%2==0)
    goto even;
else goto odd;
even:
    printf("Number is even");
    goto end;
odd: printf("Number is odd");
    goto end;
end:
   printf("\n");
}
```

Problem 1: WAP in C to convert the given binary number into decimal.

```
#include <stdio.h>
void main()
{
 int num, b_val, d_val=0, base=1, rem;
 printf("Enter a binary number (0's and 1's) \n");
 scanf("%d", &num);
 b_val = num;
 while (num>0)
 {
  rem = num%10;
  d_val = d_val + rem * base;
  num = num/10;
  base = base*2;
 }
 printf("The binary number is =%d \n", b_val);
 printf("Its decimal value is =%d \n", d_val);
}
```

output

```
Enter a binary number 0s and 1s
10101 ↵
the binary number is = 10101
Its decimal value is = 21
```

Problem 2. C Program to convert the given decimal number into binary number.

```
#include<stdio.h>
void main()
{
 long num, d_num, rem, base=1, b_num=0;
 printf("Enter a decimal number");
 scanf("%ld", &num);
 d_num = num;
 while (num>0)
 {
  rem = num%2;
  b_num = b_num + rem * base
  num = num/2;
  base = base*10;
 }
 printf("Its equivalent decimal binary number=%ld \n", b_num);
}
```

output

```
Enter a decimal number
134
its equivalent binary
number = 10000110
```

Pyramids

```
*        1        1        1              2        1        5        5
**       2 2      1 2      2 3            3 4      0 1      5 4      4 4
***      3 3 3    1 2 3    4 5 6          4 5 6    1 0 1    5 4 3    3 3 3
****     4 4 4 4  1 2 3 4  7 8 9 10       5 6 7 8  0 1 0 1  5 4 3 2  2 2 2 2
*****    5 5 5 5 5 1 2 3 4 5 11 12 13 14 15 6 7 8 9 10 1 0 1 0 1 5 4 3 2 1 1 1 1 1
  (a)      (b)      (c)          (d)          (e)      (f)      (g)      (h)
```

(a)

```
main()
{
    int i,j,n;
    printf("Enter n : ");
    scanf("%d",&n);

    for(i=1;i<=n;i++)
    {
        for(j=1;j<=i;j++)
            printf("* ");
        printf("\n");
    }
}
```

(b) print the value of i

(c) print value of j

(d) take a variable p = 1 write the printf statement as

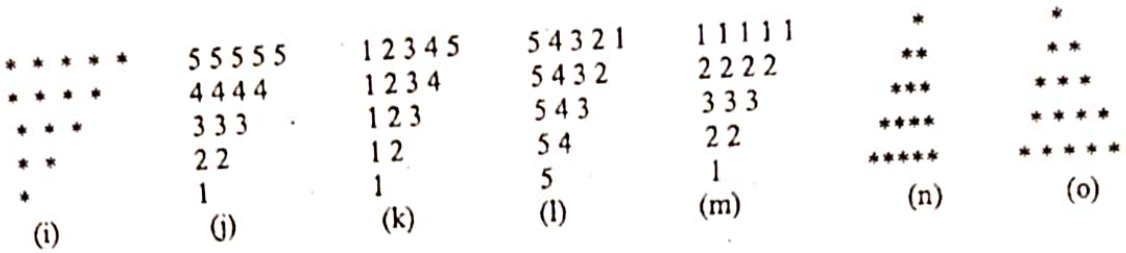
printf("%3d",p++);

(e) print the value of i+j

(f) print 1 if (i+j) is even and print 0 if (i+j) is odd.

(g) print (n+ i-j)

(h) print (n+ i-i)



(i) (j) (k) (l) (m) (n) (o)

(i)

```
for(i=n;i>=1;i--)
{
    for(j=1;j<=i;j++)
        printf("*  ");
    printf("\n");
}
```

(j) print i

(k) print j

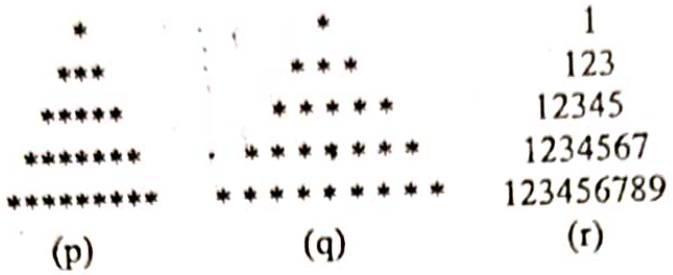
(1) print (n+ i-j)

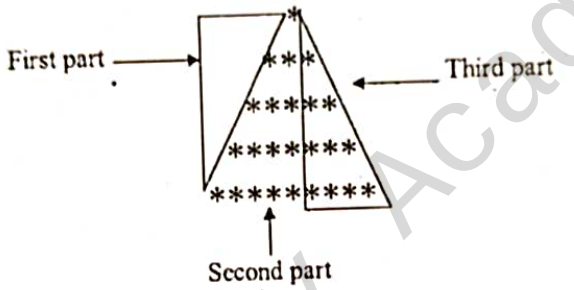
(m) (n+ i-i)

(n)

```
for(i=1;i<=n;i++)
{
    for(j=1;j<=n-i;j++)
        printf("  ");
    for(j=1;j<=i;j++)
        printf("*");
    printf("\n");
}
```

(o) The code for pyramid (0) is same as this one (n), only a space is given after star in the printf statement





(p)

```
for  (i=1;i<=n;i++)
{
    for(j=1;j<=n-i;j++)
        printf(" ");
    for(j=1;j<=i;j++)
        printf("*");
    for(j=1;j<i;j++)
        printf("*");
    printf("\n");
}
```

(q) first for loop for spaces will be from 1 to 2*(n+I-i) and there will be a space after star in the printf statements

(r) we'll initialize the value of p with 1 each time before second inner for loop; and then
print the value of p++ in the last two for loops.

3.2. Functions.

A Function is a block of code that performs a particular task. A 'C' Program consist of one or more functions. If a program has only one function then it must be the main() function.

There are following benifit of using function.

1. A ~~dif~~ Program can be divided into functions each of which perform some specific task.
2. It make code reusable
3. The program becomes easily understandable, modifiable and easy to debug and test.
4. function can be stored in library and reusability can be achieved.

3.2.1 Types of Functions.

(i) Library function.

(ii) User defined function.

(i) Library Function:

'C' has the facility to provide library functions for performing some operations. These functions are present in the C library and they are predefined. eg: → sqrt(), printf(), scanf() etc.

To use a library function we have to include corresponding header file using preprocessor directives.

sqrt() ⟶ math.h

printf(), scanf() → stdio.h

clrscr(), getch() → conio.h

Program to find square root.

Example:

```
#include<stdio.h>
#include<math.h>
void main()
{
double n,s;
printf("Enter a number");
scanf("%lf",&n);
s=sqrt(n);
printf("the square root of %lf is %lf\n",n,s);}
```

output.

```
Enter a number 16
the square root of 16 is : 4
```

(ii) User-defined functions:

Users can create their own functions for performing any specific task of the program. these type of function called User defined function.

To create and use these functions, we should know about these three things -

1. Function definition
2. function declaration.
3. function call.

Example: Program to find sum of two number using function.

```
#include<stdio.h>
int sum(int x, int y); // function declaration or Prototyping
void main()
{
int a, b, s;
printf("Enter value for a and b: ");
scanf("%d %d", &a, &b);
s=sum(a, b); // function calling
printf("sum of %d and %d is %d\n", a, b, s);
}
int sum(int x, int y) // function
{
  int z;
  z=x+y;
  return z;
}
```

1. Function Declaration/Prototyping.: A function prototype is simply declaration of function that specifies function's name, parameters and return type. A function prototype gives information to the compiler that the function may later be used in the program.

Syntax for function declaration.

returntype functionname (type1 parameter1, type2 parameter2...);

function declaration consist four part.

- returntype → int, float, char, double.
- function name → valid identifiers.
- Parameter list → Formal parameter
- termination semicolon → ;

2. Function Definition.

function definition contain the block of code to perform a specific ~~code~~ task.

Syntax:

```
return type functionname(type1 parameter1, type2 parameter2, ----)
{
    // Body of function.
}
```

function header → return type functionname(type1 parameter1, type2 parameter2, ----)

function body → { // Body of function. }

function body contain the declaration and statements necessary for performing the required task. the function body consist of

- local variable (if required)
- function statement
- return statement to return result evaluated by function (if return type is void then no return statement required).

3. Function Calling: when a function is called control of program gets transferred to the function.

Syntax:

functionname(argument1, argument2 ----);

Example: Program to find number is even or odd.

```
#include<stdio.h>
void find(int n);   // function declaration or prototype.
void main()
{
  int num;
  printf("Enter a number");
  scanf("%d",&num);
  find(num);        // function calling
}
void find(int n)    // function definition.
{
if(n%2==0)
    printf("%d is even",n);
else
    printf("%d is odd",n);
}
```

3.2.1. Return Statement.

The return statement terminates the execution of a function and return the value to calling function. the program control is transfered to the calling function after the return statement.

Syntax:

```
return (expression);
```

eg.

```
return a;
return (a+b);
return;
return (x+y*z);
return (3*sum(a,b));
```

the return is keywords.

3.2.3. Function Arguments.

The calling function send some values to called function for communication these value are called argument or Parameters

There are two types of argument.

1. Actual Arguments
2. Formal Arguments

Actual Arguments and Formal Arguments.

The variable declare in the function prototype or function definition are known as Formal Arguments and value passed to called function from calling function, the four Argument which are mentioned in function call are known as Actual Arguments.

Syntax:

```
#include<stdio.h>
return_type fun_name(arguments);
void main()
{
fun_name(arguments)
}
return_type fun_name(arguments)
{
 - - -
 - - -
}
```

Formal Arguments

Actual Arguments

Example:

```
#include<stdio.h>
void main();
void func(int a, int b);
void main()
 {
int a=6 b=3;
 func(a, b);
func(15, 4);
func(a+b, a-b);
}
void func(int a, int b)
{
 printf("a=%d", b=%d\n"a,b)
}
```

output

```
a=6  b=3
a=15  b=4
a=9  b=3
```

3.24 Types of User defined function: (classification of User defined function)

Function can be classified into four categories on the basis of argument and return type.

1. Function with no argument no return value.
2. Function with no argument and a return value.
3. Function with argument and no return value.
4. Function with argument and a return value.

Function with no argument no return type:

Functions that have no argument and no return type/value is written as.

```
#include<stdio.h>
void func (void);
void main ( )
{
  - - -
  - - -
  func( );
}
void func (void)
{
 - -
 - -
}
```

Example:

```
#include<stdio.h>
void valu(void);
void main( )
{
 valu( );
}
void valu(void)
{
float p, t, r, SI;
printf("Enter Principle Amount, time, and rate );
Scanf( "%f %f %f", &p, &t, &r);
SI = (p*t*r) / 100;
printf("Simple Interest = %f, SI);
}
```

```
Enter Principle Amount,
time and rate 1200 ↵
2 ↵
5.4 ↵
Simple Interest = 129.600006.
```

Function with no argument and a return value :

There could be situation where we need to design a function that may not tak any argument but return a value.

```
#include<stdio.h>
int func (void);
void main()
{
 ----
 ----
 func();
}
int func(void)
{
  ---
  ---
 return(expression);
}
```

Example:

```
#include<stdio.h>
#include<math.h>
int sum( );
void main( )
{
 int num;
 num = sum( );
 printf("sum of squar root of given two number=%d",num);
 }
int sum()
 {
 int a, b, sum;
 Printf(" Enter two number");
 scanf("%d,%d", &a, &b);
 sum = sqrt(a) + sqrt(b);
 return(sum);
 }
```

output

```
Enter two number: 25 49↵

sum of squre root of given two number
 = 12.
```

Function with arguments but no return value:

These type of functions have arguments, hence calling function can send data to called function but called function does not return any value.

```
#include<stdio.h>
void func(int, int);
void main()
{
 func(a, b);
}
void func(int x, int y)
{
 statement;
}
```

Example: // Area of square:

```
#include<stdio.h>
void area(int side);
void main()
{
 int side;
 printf("Enter the side of square:");
 scanf("%d", &side);
 area(side);
}
void area(int side)
{
 int area;
 area = side * side;
 printf("Area of square=%d", area);
}
```

output.

```
Enter the side of square: 8↵
Area of square = 64
```

Function with Argument and return value;

these type of functions have argument, so calling function send data to the called function, it can also return a value to calling function. using return statement.

```
#include <stdio.h>
int func(int int);
void main()
{
  - -
  - -
 func(a, b);
}
int func (int x, int y)
{
  - - -
  - - -
 return (expression);
}
```

Example: // sum of digit.

```
#include<stdio.h>
int sum(int n);
void main()
{
 int num;
printf("Enter the number:");
scanf("%d", &num);
printf("sum of digit = %d", sum(num));
}
int sum (int n)
{
 int i, sum=0, rem;
while(n>0)
 {
  rem = n%10;
  sum = sum + rem;
  n = n/10;
 }
return (sum);
}
```

Output

```
Enter the number: 235
sum of digit = 10
```

3.2.5: Local, Global and Static Variable:

Local variable: The variable that are declare and defined within the body of function or a block are local to that function or block only and are called local variable.

e.g

```
fun_1( )
{
  int a, b; //local
  ---
  ---
}
```

Syntax:

```
fun_2( )
{
 int a=2, b=4; //local
  ---
  ---
}
fun_3( )
{
 int a=15, b=20; //local
 ---
 ---
}
```

Here a=2, b=4 are local to function fun_2() and a=15, b=20 are local to fun_3().

Global Variable: the variable that are defined outside of any function are called global variable. they are not limited to any function. Any function can access and modify global variable.

```
#include<stdio.h>
int a, b; //global variable
void main()
{
 ---
 ---
}
```

```
#include<stdio.h>
void fun1(void);
void fun2(void);
int a=3, b=6;
void main()
{
 printf("a=%d, b=%d", a, b);
 fun1();
 fun2();
}
```

```
void fun1( )
{ printf("a=%d, b=%d", a, b);
}
void fun2()
{ int a=8;
 printf("a=%d, b=%d", a, b)
}
```

output

a=3, b=6
a=3, b=6
a=8, b=6

Static Variable.

Static variables are declared by writing keyword 'static' as:

static datatype variable_name.

eg static int a;

A static variable is initialized only once and the value of static variable is retained betn function calls. if static variable not initalized then it is automatically initialized to 0.

eg.

```
#include<stdio.h>
void fun(void)
void main()
  {
   fun();
   fun();
   fun();
  }
void fun()
{
int a=10
static int b=10
printf("a=%d, b=%d", a, b);

a++;
 b++;
}
```

output.

```
a=10  b=10
a=10  b=11
a=10  b=12.
```

3.2.6 Call by Value and Call by Reference.

1. Call by Value:

This method copies the actual value of a argument into the formal parameter of function. In this case changes made to the parameter inside the function have no effect on the argument

by default c Programming uses call by value to pass the argument

```
// Program to swapping of two numbers using Call by Value.
#include <stdio.h>
void swap(int x, int y);
void main()
{
 int a=10, b=20;
 printf("Before swapping value of a=%d, b=%d", a, b);
 swap(a, b);
}
void swap(int x, int y)
{
int temp;
temp=x;
x= y;
y= temp;
printf("After the swapping value of a=%d, b=%d", x, y);
}
```

2. Call by References.

This method copies the address of an argument into formal parameter. Inside the function the address ~~of~~ an is used argument ~~in~~ to access the actual argument used in the call. this means that changes made to the parameter affect argument.

// Program to swap two numbers using call by references.

```
#include<stdio.h>
void swap(int *x, int *y);
void main()
{
int a=10, b=20;
printf("Before swapping a=%d, b=%d", a, b);
swap(&a, &b);
printf("After swapping a=%d, b=%d", a, b)
}
void swap(int *x, int *y)
{
int temp;
temp= *x;
*x= *y;
*y= temp;
}
```

3.2.7. Recursion:

Recursion is a process of repeating item in self-similar way. In programming, if a program allows you to call a function inside the same function then it is called recursive call of function.

In Recursion calling function and called function are same.

Syntax:

```
main()
{
---
---
rec();
}
rec ()
{
---
---
rec();
}
```

Recursive Call

There should be a terminating condition to stop recursion, otherwise rec() will keep calling itself.

```
rec()
{
---
if( )  //termination condition
---
rec ();
}
```

Recursive function are very helpfull to solve problem. such as:

(i) Factorial.
(ii) Power
(iii) Fibonacci numbers
(iv) Tower of Hanoi.

2. Call by References.

This method copies the address of an argument into formal parameter. Inside the function the address ~~of~~ is used ~~argument~~ in to access the actual argument used in the call. this means that changes made to the parameter affect argument.

// Program to swap two numbers using call by references.

```
#include<stdio.h>
void swap(int *x, int *y);
void main()
{
int a=10, b=20;
printf("Before swapping a=%d, b=%d", a, b);
swap(&a, &b);
printf("After swapping a=%d, b=%d", a, b)
}
void swap(int *x, int *y)
{
 int temp;
 temp= *x;
*x= *y;
*y= temp;
}
```

(i) Program to find factorial of any number using function. (Recursive method)

```
#include<stdio.h>
long fact(int n);
void main()
{
int num;
Printf("Enter a number:");
Scanf("%d", &num);
printf("Factorial of given number is = %ld", fact(num));
}
long fact(int n)
{
  if(n==0)
    return(1)
  else
    return(n*fact(n-1));
}
```

(ii) Program to find power using recursion.

```
#include<stdio.h>
int power(int base, int a);
void main()
{
int base, a, result;
printf("Enter base and Power positive integer:");
scanf("%d"%d"", &base, &a);
result= power(base, a);
printf("%d^%d=%d", base, a, result);
}
```

```
int power(int base, int a)
{
if(a!=0)
    return(base*power(base,a-1));
else
    return 1;
}
```

Program to generate fibonacci series using recursion:

```
#include<stdio.h>
int fib (int n)
void main( )
{
  int term, i;
  printf("Enter number of terms : ");
  scanf("%d", &term);
  for(i=0; i< term ; i++)
        printf( "%d ", fib(i));
  printf("\n);
}
int fib (int n)
{
 if ( n==0 ||n==1)
       return 1;
 else
    return (fib(n-1) + fib(n-2));
}.
```

Important Questions:

1. Differentiate between while and do-while loop.

2. Take the three digit number from the user then write a program to check entered number is palindrome or not.

4. Write a program in C to generate the Fibonacci series up to the last Fibonacci number less than 100. Also finds the sum of all Fibonacci numbers and total count of all Fibonacci numbers.

5. Write a program in C to print the following pattern:

2 3 4 5 6 7

3 4 5 6 7

4 5 6 7

5 6 7

6 7

7

6. Write the syntax format for while, do while and for loops. Write a program in C to multiply a matrix of dimension 4*4 and store the result in another matrix.

7. Write a program in C to print following pattern with appropriate comments:

10

9 8

7 6 5

4 3 2 1

8. Differentiate between call by value and call by reference. Write a program in that computes the area and circumference of a circle with radius taken as input using call by reference in functions

9. Write a recursive function in C, which takes an input from user to calculate a factorial using the recursion concept

10. What do you mean by call by value and call by reference? Write an algorithm for swapping two numbers using call by reference technique. Also write a C program for the above stated algorithm

11. What is a function? Why programmers use functions in code? While executing a function, how the values are passed between calling and called environment?

Unit: IV

Array and Basic Algorithm.

4.1 Array:

An array is a collection of similar type of data item and each data item called an element of the Array. Data items stored in array at contiguous memory location. The Data type of array may be any valid data type like int, char or float.

Consider a situation when we want to store age of 100 employee

* one way is declare 100 variable, store value to each variable and Display them. This is very difficult when no. of variable is very large.
* another way is declare an Array of size 100 of int type eg: `int age[100];` this is very easy to access and store the data item.

The individual elements of this array are.

age[0], age[1], age[2] . . . . . age[99]

in C Programing index will start from 0. so age[0] is the first element, age[1] is the second element.

Advantage of Array:

1. Code optimization
2. Ease of traversing
3. Ease of sorting
4. Random Acess

Disadvantage of Array:

1. Fixed size of array.

4.2 Types of array.

1. One Dimensional Array (1D)
2. Two dimensional array (2D)
3. Multi dimensional Array

4.2.1 One Dimensional Array:

(1) **Declaration**: the syntax for declaration 1D-Array is

```
data_type array_name[size];
```

- data_type → valid data type, eg int, char, float.
- array_name → valid Identifiers
- size → subscript for size of array.

eg. `int age[100];` → can hold 100 element of integer type

`float sal[15];` → can hold 15 element of float type

`char grade[20];` → can hold 20 character.

the individual element of above arrays are:

age[0], age[1] ------ age[99]

sal[0] sal[1] ------ sal[14]

grade[0] grade[1] ------ grade[19]

when array is declare compiler allocates space in memory to hold all the element.

(ii) Accessing 1-D Array:

The elements of an array can be accessed by specifying the array name followed by subscript in bracket.

for int arr[5]

the element of this array are

arr[0], arr[1], arr[2] arr[3] arr[4]

subscript can be any expression that yields an integer value. It can be any integer constant, variable, int expression.

eg. arr[3], arr[i], arr[i+j], arr[2*j]

(iii) Processing of 1-D Array:

subbase arr[10] is an array of int type.

(a) Reading value in arr[10]

```
for(i=0; i<10; i++
  {
    scanf("%d", &arr[i]);
  }
```

(b) Displaying value of arr[10]

```
for(i=0; i<10; i++)
  {
    printf("%d", arr[i]);
  }
```

(c) Adding all element of arr[10]

```
sum=0
for(i=0; i<10; i++)
 {
   sum = sum + arr[i];
 }
```

(iv) Initialization of 1-D Array

Syntax:

```
data_type arr_name[size] = { value1, value2, valueN };
```

e.g.

```
int marks[5] = { 50, 85, 70, 65, 95 }

int sal[] = { 25.5, 38.5, 24.7 }
```

Example Program to take input into array and Display the element.

```
#include <stdio.h>
Void main()
{
 int value[5], i;
 printf(" Enter 5 integer");
 //taking input and storing
 for(i=0; i<5; i++)
   { scanf("%d", &value[i]);
   }
printf("Displaying integers");
// Printing elements of an array
for(i=0 ; i<5; i++)
   {
    printf("%d \n", value[i]);
   }
}
```

output

```
Enter five Integer
10
15
20
12
16
Displaying integers
10
15
20
12
16
```

// Program to find the average of n numbers using arrays

```
#include <stdio.h>
void main()
{
   int marks[100], i, n, sum = 0, average;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("Enter number %d: ", i);
      scanf("%d", &marks[i]);
      sum += marks[i];
   }
   average = sum/n;
   printf("Average = %d", average);
}
```

//Program to search for an item in the array*/

```
# include<stdio. h>
#define SIZE 10
void main ( )
{
int i,arr[SIZE]={23,12,56,98,76,14,65,11,19,45};
int item;
printf ("Enter the item to be searched ") ;
scanf("%d", &item);
for(i=0; i<SIZE; i++)
{
if(item==arr[i])
{
printf ("%d found at position %d\n", item, i);
break;
}
if (i= =SIZE)
printf("Item %d not found in array\n" , item) ;
}
```

4.2.2 Passing Array to Functions

a) Passing Individual Array Element to a function: Pass the Individual array elemnt to function like other simple variable.

// Program to pass Array element to a function to check elemnt is even or odd.

```
#include<stdio.h>
void check(int num);
void main()
 {
  int arr[10],i;
  printf("Enter the ten elemnt to array:");
  for(i=0; i<10; i++)
   {
     scanf("%d", &arr[i]);
     check(arr[i]);
   }
 }
void check(int num)
{
 if(num%2==0)
     printf("%d is even\n", num)
 else
     printf("%d is odd\n", num)
 }
```

Output:

```
Enter the ten elemnt to array
15 ↵
15 is odd
16 ↵
16 is even
45 ↵
45 is odd.
```

(b) Passing whole 1-D array to a function.

we can pass whole array as an actual argument to a function. the corresponding formal argument should be declared as an array variable of same data type.

#program to calculate the sum of array element by passing to a function.

```
#include<stdio.h>
float sum(float age[]);
void main()
 {
float result, age[ ] = {23.4, 55, 22.6, 3, 40.5, 18};
  result = sum(age);
  printf( "Result = %.2f", result);
 }
float sum(float age[ ])
{
float sum = 0.0;
for(int i=0; i<6 ; i++)
    {sum = sum + age[i];
    }
 return sum;
}
```

output

Result = 162.50

4.2.3. Two Dimentional Array (matrices)

Two Dimentional Array can be defined as array of array. The 2D Array is organized as matrices which can represented as the collection of rows and columns.

Declaration of 2D Array:

The syntax to declare 2-D Array is:

data_type array_name [row_size] [column_size];

eg. int arr[4][3]
- No. of column.
- No. of rows.

the starting element of this array is arr[0][0] and last element of this array is arr[3][2]

Hence total No. of elemnt in ~~This~~ array is ~~4×3~~ = 12.

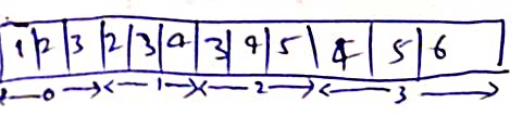
	0	1	2
0	arr[0][0]	arr[0][1]	arr[0][2]
1	arr[1][0]	arr[1][1]	arr[1][2]
2	arr[2][0]	arr[2][1]	arr[2][2]
3	arr[3][0]	arr[3][1]	arr[3][2]

Initialization of 2D-Array.

int arr[4][3] = {{1,2,3},{2,3,4},{3,4,5},{4,5,6}}

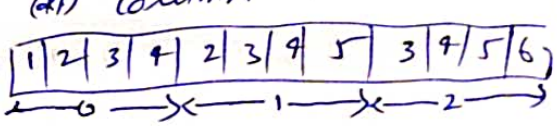
Representation:

(i) row major.



	0	1	2
0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

(ii) column major.



Processing of 2-D Array.

```
int arr[4][5]
```

(i) Reading value in arr

```
for(i=0; i<4; i++)
    for(j=0; j<5; j++)
        scanf("%d", &arr[i][j]);
```

2. Displaying output to screen.

```
for(i=0; i<4; i++)
{
    for(j=0; j<5; j++)
        printf("%d", arr[i][j]);
    printf("\n");
}
```

Example 1. // Program to take input and display matrix.

```
#include<stdio.h>
#define row 3
#define col 4
void main()
{
int mat[row][col], i, j;
printf("Enter the elemt for matrix of 3 row and 4 column");
for(i=0; i<3; i++)
    for(j=0; j<4; j++)
        scanf("%d", &mat[i][j]);
printf("The matrix that you have entered");
for(i=0; i<3; i++)
{
    for(j=0; j<4; j++)
        printf("%d", mat[i][j]);
    printf("\n");
}
```

output

```
Enter the elemt for matrix 3 row and
4 column:
2 3 4 7
8 5 1 9
1 8 2 5
The matrix that you have entered:
2 3 4 7
8 5 1 9
1 8 2 5
```

Example 2. // Programs for Addition of two matrices.

```
#include<stdio.h>
#define row 3
#define col 4
void main()
{
int i, j, mat1[row][col], mat2[row][col], mat3[row][col];
printf("Enter matrix for elements for first matrix");
for(i=0; i<row; i++)
    for(j=0; j<col; j++)
        scanf("%d", mat1[i][j]);
printf("Enter elemnt for second matrix");
for(i=0; i<row; i++)
    for(j=0; j<col; j++)
        scanf("%d", mat2[i][j]);
// Addition.
for(i=0; i<row; i++)
    for(j=0; j<col; j++)
        mat3[i][j] = mat1[i][j] + mat2[i][j];
printf("The resultant matrix are:");
for(i=0; i<row; i++)
  {
    for(j=0; j<col; j++)
        printf("%d", mat3[i][j]);
    printf("\n");
  }
}
```

output

```
Enter elemnt for first matrix
1 2 4 8
5 6 7 8
2 3 1 4
Enter elemnt for second matrix
2 5 4 2
1 5 2 6
9 4 7 2
the resultant matrix are
3 7 12 6
6 11 9 14
12 6 8 6
```

```
      Program for multiplication of two matrices*/
#include<stdio.h>
#define ROW1 3
#define COL1 4
#define ROW2 COL1
#define COL2 2
main( )
{
    int mat1[ROW1][COL1],mat2[ROW2][COL2],mat3[ROW1][COL2];
    int i,j,k;

    printf("Enter matrix mat1(%dx%d)row-wise :\n",ROW1,COL1);
    for(i=0;i<ROW1;i++)
        for(j=0;j<COL1;j++)
            scanf("%d",&mat1[i][j]);

    printf("Enter matrix mat2(%dx%d)row-wise :\n",ROW2,COL2);


    for(i=0;i<ROW2;i++)
        for(j=0;j<COL2;j++)
            scanf("%d",&mat2[i][j]);

    /*Multiplication */
    for(i=0;i<ROW1;i++)
        for(j=0;j<COL2;j++)
        {
            mat3[i][j]=0;
            for(k=0;k<COL1;k++)
                mat3[i][j]+=mat1[i][k]*mat2[k][j];
        }

    printf("The Resultant matrix mat3 is :\n");
    for(i=0;i<ROW1;i++)
    {
        for(j=0;j<COL2;j++)
            printf("%5d",mat3[i][j]);
        printf("\n");
    }
}
```

Output:

```
Enter matrix mat1(3x4)row-wise -
    2 1 4 3
    5 2 7 1
    3 1 4 2
Enter matrix mat2(4x2)row-wise -
    1 2
    3 4
    2 5
    6 2
The Resultant matrix mat3 is -
    31   34
    31   55
    26   34
```

```
        Program to find the tranpose of matrix.*/
#include<stdio.h>
#define ROW 3
#define COL 4
main()
{
    int mat1[ROW][COL],mat2[COL][ROW],i,j;
    printf("Enter matrix mat1(%dx%d) row-wise : \n",ROW,COL);

    for(i=0;i<ROW;i++)
        for(j=0;j<COL;j++)
            scanf("%d",&mat1[i][j]);

    for(i=0;i<COL;i++)
        for(j=0;j<ROW;j++)
            mat2[i][j]=mat1[j][i];

    printf("Tranpose of matrix is:\n");
    for(i=0;i<COL;i++)
    {
        for(j=0;j<ROW;j++)
            printf("%5d",mat2[i][j]);
        printf("\n");
    }
}
```

Output:

Enter matrix mat1(3x4) row-wise-

3 2 1 5
6 5 8 2
9 3 4 1

Tranpose of matrix is-

3 6 9
2 5 3
1 8 4
5 2 1

4.2.4. Array with more than two Dimensions (3-D) (Multi Dimentional)

Three dimension array is an array of 2-D array.

Syntax.

```
int arr[2][4][3]
```

(row and column of 2-D array.)

Two 2-D array and each of that 2-D array has 4 row and 3 colomn.

[0]

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]
[3][0]	[3][1]	[3][2]

[1]

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]
[3][0]	[3][1]	[3][2]

the individual array elmut are

arr[0][0][0], arr[0][0][1], arr[0][0][2] .... arr[0][3][2]

arr[1][0][0]. arr[1][0][1] arr[1][0][2] - - - . arr[1][3][2]

Total No of elments are.

$2 \times 4 \times 3 = 24$.

4.3 Basic Algorithms

4.3.1 Searching &Basic Sorting Algorithms

Searching: Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

i. Linear
ii. Binary

Sorting Algorithms: A Sorting Algorithm is used to rearrange a given array or list elements in ascending or descending order. We will discuss following sorting algorithms

i. Bubble
ii. Insertion
iii. Selection

4.3.1.1 Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.

Algo: Linear Search (Array A, Value x)

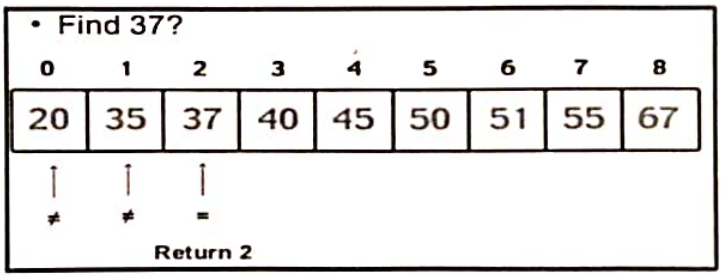
Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2



Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

```
// C Program to Implement Linear Search
#include <stdio.h>
void main()
{
 int array[10], search, i;
 printf("Enter number of elements in array\n");
   for (i = 0; i < 10; i++)
        scanf("%d", &array[i]);
 printf("Enter a number to search\n");
 scanf("%d", &search);

for (i = 0; i< 10; i++)
 {
  if (array[i] == search)
  {
   printf("%d is present at location %d.\n", search, i);
   break;
  }
 }
 if (i == 10)
  printf("%d isn't present in the array.\n", search);
}
```

Output

```
Enter number of elements in array
12
5
7
18
45
34
21
61
9
10
Enter a number to search
21
21 is present at location 6.
```

Complexity of algorithm

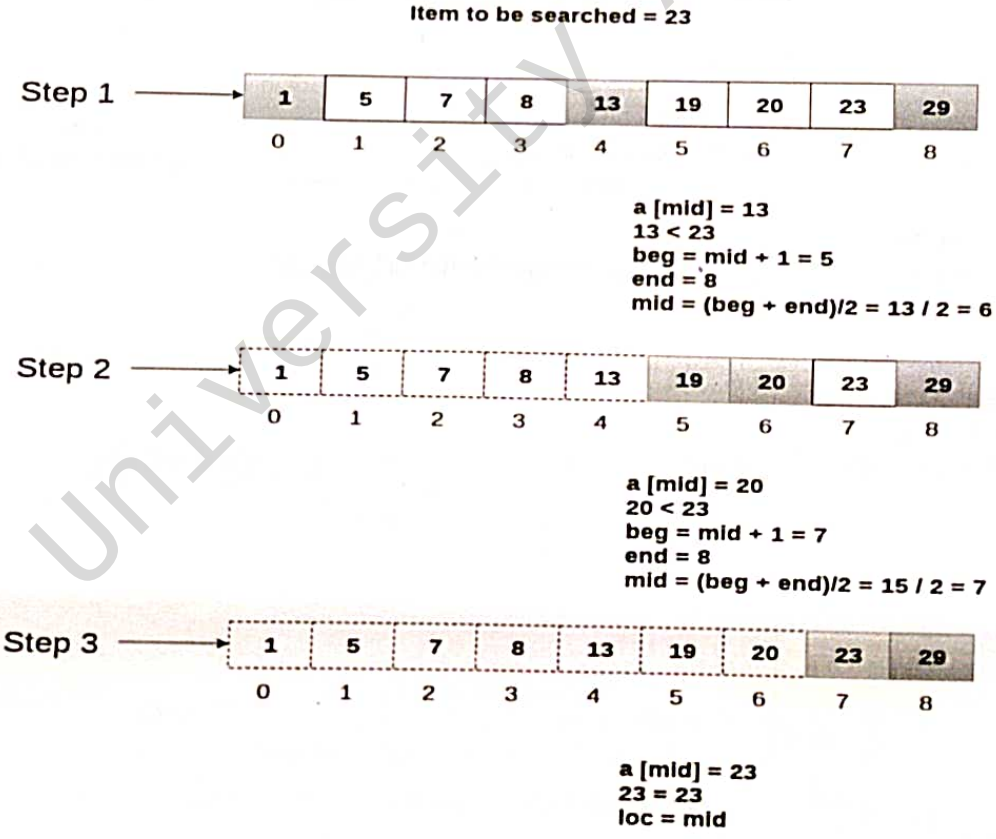
Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

4.3.1.2 Binary Search Algorithm

Binary Search is applied on the sorted array or list of large size. It's time complexity of O(log n) makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Following are the steps of implementation that we will be following:

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
        - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
2. When a match is found, return the index of the element matched.
3. If no match is found, then return -1



```
#include <stdio.h>
void main()
{
        int c, first, last, middle, n, search, array[100];
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d integers\n", n);
                for (c = 0; c < n; c++)
                        scanf("%d", &array[c]);
        printf("Enter value to find\n");
        scanf("%d", &search);
        first = 0;
        last = n - 1;
        middle = (first+last)/2;
        while (first <= last)
                {
                if (array[middle] < search)
                        first = middle + 1;
                else if (array[middle] == search)
                        {
                        printf("%d found at location %d.\n", search, middle);
                        break;
                        }
                 else
                        last = middle - 1;
                middle = (first + last)/2;
                }
        if (first > last)
         printf("Not found! %d isn't present in the list.\n", search);
}
```

Output:

```
Enter number of elements
5
Enter 5 integers
45
56
67
78
89
Enter value to find
67
67 found at location 2.
```

Complexity

N	Performance	Complexity
1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)
4	Worst case space complexity	O(1)

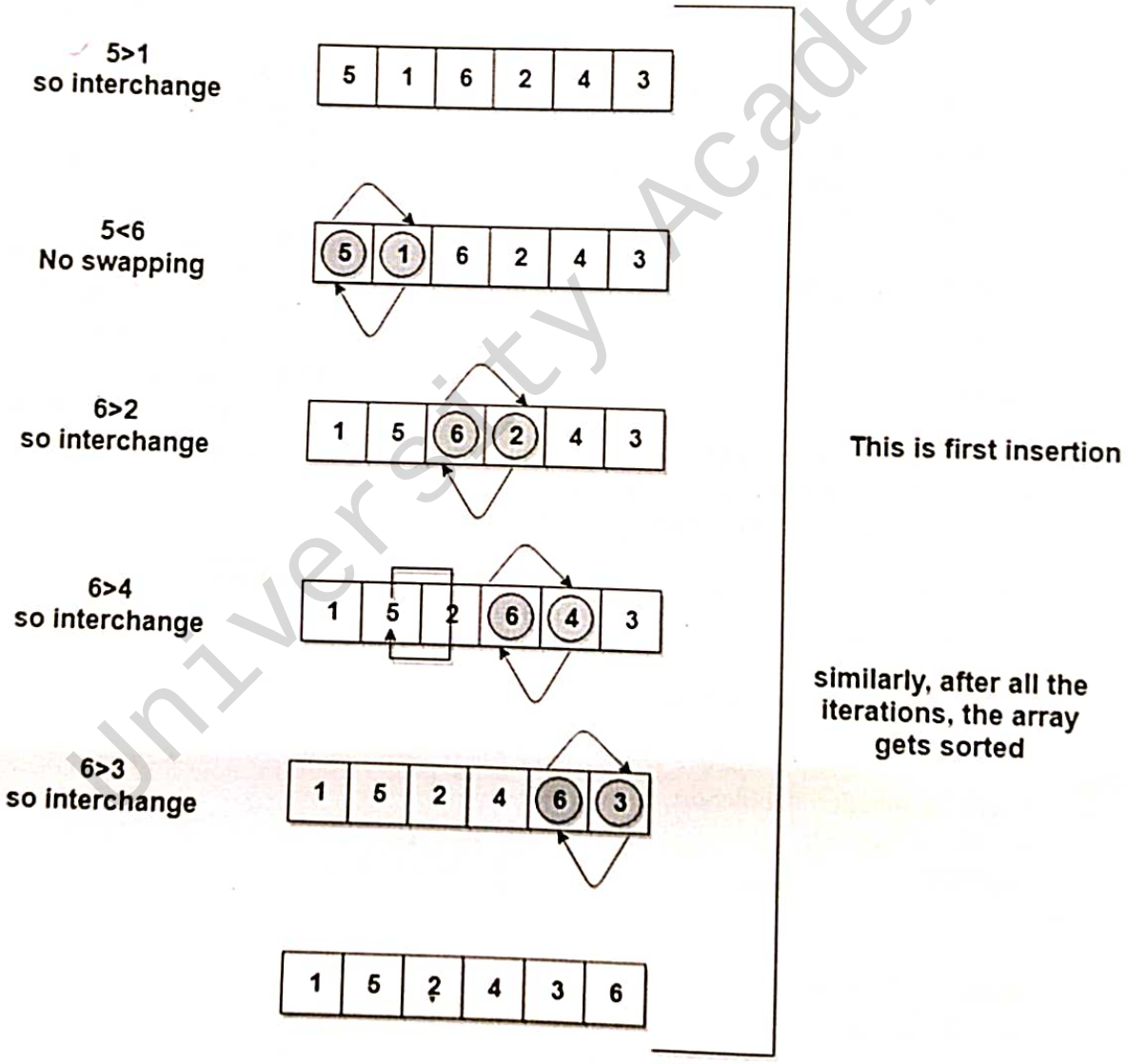
4.3.1.3 Bubble Sort Algorithm

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1.**

Let's consider an array with values {5, 1, 6, 2, 4, 3}



Program: Simple C program for bubble sort

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = 0; j < n-i-1; j++)
      {
         if( arr[j] > arr[j+1])
         {
            // swap the elements
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }

   // print the sorted array
   printf("Sorted Array: ");
   for(i = 0; i < n; i++)
   {
      printf("%d  ", arr[i]);
   }
}
int main()
{
   int arr[100], i, n, step, temp;
   // ask user for number of elements to be sorted
   printf("Enter the number of elements to be sorted: ");
   scanf("%d", &n);
   // input elements if the array
   for(i = 0; i < n; i++)
   {
      printf("Enter element no. %d: ", i+1);
      scanf("%d", &arr[i]);
   }
   // call the function bubbleSort
   bubbleSort(arr, n);
    return 0;
}
```

Output:

```
Enter the number of elements to be sorted:
5
Enter element no. 1: 67
Enter element no. 2: 78
Enter element no. 3: 79
Enter element no. 4: 8
Enter element no. 5: 5
Sorted Array: 5  8  67  78  79
```

Complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity: O(n2)
- Best Case Time Complexity: O(n)
- Average Time Complexity: O(n2)
- Space Complexity: O(1)

4.3.1.4 Insertion Sort Algorithm:

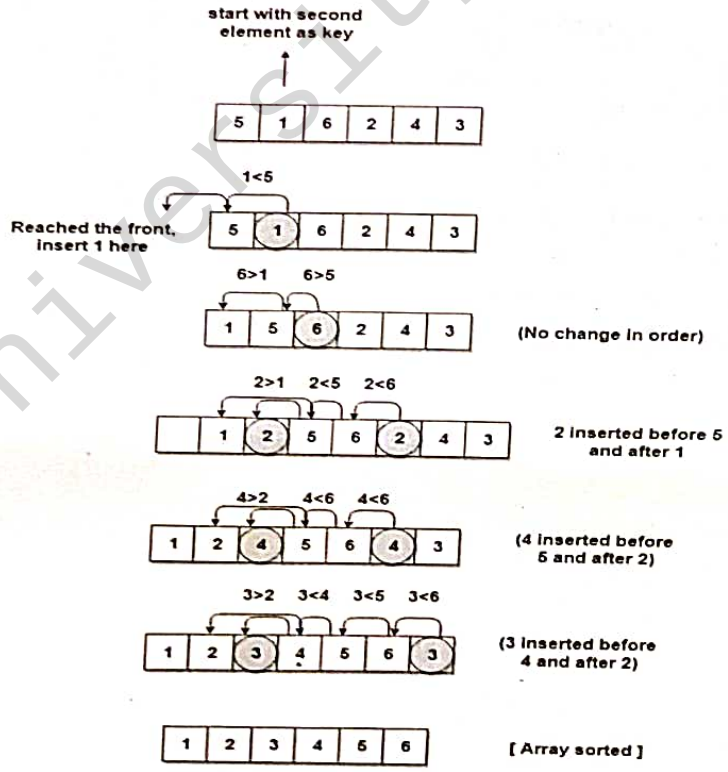
Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers. If I give you another card, and ask you to insert the card in just the right position, so that the cards in your hand are still sorted. What will you do?

Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will insert the card there.

Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We compare the key element with the element(s) before it, in this case, element at index 0:
    - If the key element is less than the first element, we insert the key element before the first element.
    - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Let's consider an array with values {5, 1, 6, 2, 4, 3}



Program: C program to implement Insertion Sort

```
#include<stdio.h>
void main()
{
  int i, j, count, temp, number[25];
  printf("How many numbers u are going to enter?: ");
  scanf("%d",&count);

  printf("Enter %d elements: ", count);
  // This loop would store the input numbers in array
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);

  // Implementation of insertion sort algorithm
  for(i=1;i<count;i++){
    temp=number[i];
    j=i-1;
    while((temp<number[j])&&(j>=0)){
      number[j+1]=number[j];
      j=j-1;
    }
    number[j+1]=temp;
  }
  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
}
```

Output:

```
How many numbers u are going to enter?: 5
Enter 5 elements: 34
43
23
12
54
Order of Sorted elements:  12 23 34 43 54
```

Complexity

- Worst Case Time Complexity: O(n2)
- Best Case Time Complexity: O(n)
- Average Time Complexity: O(n2)
- Space Complexity: O(1)

4.3.1.5 Selection Sort Algorithm:

This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.

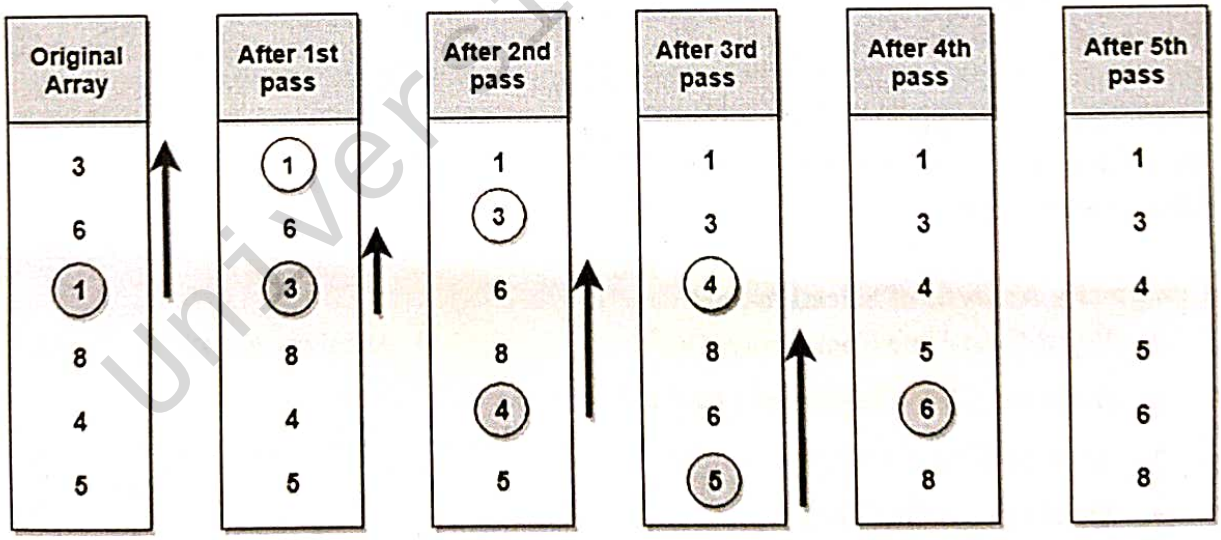
It is called selection sort because it repeatedly selects the next-smallest element and swaps it into the right place.

Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Let's consider an array with values {3, 6, 1, 8, 4, 5}

Below, we have a pictorial representation of how selection sort will sort the given array.



Program: C program implementing Selection Sort

```
#include <stdio.h>
void main()
{
int a[100], n, i, j, position, swap;
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d Numbers\n", n);
for (i = 0; i < n; i++)
scanf("%d", &a[i]);
for(i = 0; i < n - 1; i++)
{
position=i;
for(j = i + 1; j < n; j++)
{
if(a[position] > a[j])
position=j;
}
if(position != i)
{
swap=a[i];
a[i]=a[position];
a[position]=swap;
}
}
printf("Sorted Array\n");
for(i = 0; i < n; i++)
printf("%d\n", a[i]);
}
```

Output

```
Enter number of elements
5
Enter 5 Numbers
34
54
21
23
67
Sorted Array
21
23
34
54
67
```

Complexity Analysis of Selection Sort

- Worst Case Time Complexity :$O(n^2)$
- Best Case Time Complexity : $O(n^2)$
- Average Time Complexity :$O(n^2)$
- Space Complexity: O(1)

4.3.2 Roots of a Quadratic Equation

The standard form of a quadratic equation is:

$ax^2 + bx + c = 0$, where

a, b and c are real numbers and

a != 0

The term b^2-4ac is known as the discriminant of a quadratic equation. It tells the nature of the roots.

The nature of the roots depends on the Discriminant (D) where D is.

- If D > 0, the roots are real and distinct (unequal)
- If D = 0, the roots are real and equal.
- If D < 0, the roots are real and imaginary.

$$root1 = \frac{-b + \sqrt{(b^2 - 4ac)}}{2a}$$

If the discriminant > 0,

$$root2 = \frac{-b - \sqrt{(b^2 - 4ac)}}{2a}$$

If the discriminant = 0, $root1 = root2 = \frac{-b}{2a}$

$$root1 = \frac{-b}{2a} + \frac{i\sqrt{-(b^2 - 4ac)}}{2a}$$

If the discriminant < 0,

$$root2 = \frac{-b}{2a} - \frac{i\sqrt{-(b^2 - 4ac)}}{2a}$$

Program: Program to Find Roots of a Quadratic Equation

```
#include <math.h>
#include <stdio.h>
int main() {
   double a, b, c, discriminant, root1, root2, realPart, imagPart;
   printf("Enter coefficients a, b and c: ");
   scanf("%lf %lf %lf", &a, &b, &c);
   discriminant = b * b - 4 * a * c;
   // condition for real and different roots
   if (discriminant > 0) {
      root1 = (-b + sqrt(discriminant)) / (2 * a);
      root2 = (-b - sqrt(discriminant)) / (2 * a);
      printf("root1 = %.2lf and root2 = %.2lf", root1, root2);
   }
   // condition for real and equal roots
   else if (discriminant == 0) {
      root1 = root2 = -b / (2 * a);
      printf("root1 = root2 = %.2lf;", root1);
   }
   // if roots are not real
   else {
      realPart = -b / (2 * a);
      imagPart = sqrt(-discriminant) / (2 * a);
      printf("root1 = %.2lf+%.2lfi and root2 = %.2f-%.2fi", realPart, imagPart, realPart,
imagPart);
   }
   return 0;
}
```

Output

```
Enter coefficients a, b and c: 2.3
4
5.6
root1 = -0.87+1.30i and root2 = -0.87-1.30i
```

4.3.3 Notion of order of complexity

While analysing an algorithm, we mostly consider **time complexity and space complexity**. Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Usually, the complexity required by an algorithm falls under three types –

- Best Case – Minimum time required for program execution.
- Average Case – Average time required for program execution.
- Worst Case – Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- **O Notation:** The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.
- **Ω Notation:** The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best-case time complexity or the best amount of time an algorithm can possibly take to complete.
- **θ Notation:** The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

Following is a list of some common asymptotic notations –

constant	O(1)
logarithmic	O(log n)
linear	O(n)
n log n	O(n log n)
quadratic	$O(n^2)$
cubic	$O(n^3)$
polynomial	$n^{O(1)}$
exponential	$2^{O(n)}$

4.4 String:

String is a array of char type. A character array is a string if it ends with a null character ('\0').

A string constant is a sequence of characters enclosed in double quotes. eg:

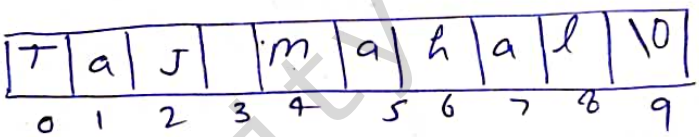
"V"

"Taj Mahal"

"2345"

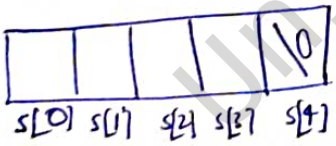
"My name is Sandeep"

String is stored in memory terminated by a null character ('\0').

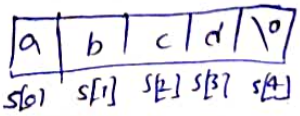


Declaration and Initialization of string.

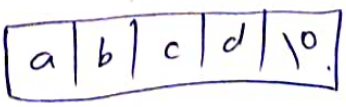
char s[5];



char s[ ] = "abcd";



char s[5] = {'a', 'b', 'c', 'd', '\0'};



Read string from the user:

(i) using scanf() function: the scanf function read the sequenced characters until it encounters whitespace (space, newline, tab etc):

e.g.
```
#include <stdio.h>
void main()
 {
 char name[30];
 printf("Enter Name:");
 scanf("%s", name);
 printf("your Name is:%s", name);
}
```

output
```
Enter Name: Sandeep Vishwakarma.
your name is Sandeep.
```

Note:. only "Sandeep" was stored in the string because there was a space after "Sandeep".

(ii) using gets() and puts(): the gets() function enables the user to enter some character followed by the enter key. all the character entered by the user get stored in a chacter array

```
#include<stdio.h>
void main()
 {
 char name[30];
printf("Enter name:");
gets(name);
printf("Entered name is:");
puts(name);
 }
```

Output
```
Enter name: Sandeep Vishwakarma
Entered name is Sandeep Vishwakarma.
```

C String Functions

C supports a wide range of functions that manipulate null-terminated strings

Sr. No.	Function & Purpose	
1	**strcpy(s1, s2);**	Copies string s2 into string s1.
2	**strcat(s1, s2);**	Concatenates string s2 onto the end of string s1.
3	**strlen(s1);**	Returns the length of string s1.
4	**strcmp(s1, s2);**	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);**	Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**	Returns a pointer to the first occurrence of string s2 in string s1.

```
#include <stdio.h>
#include <string.h>

int main () {

  char str1[12] = "Hello";
  char str2[12] = "World";
  char str3[12];
  int  len ;

  /* copy str1 into str3 */
  strcpy(str3, str1);
  printf("strcpy( str3, str1) :  %s\n", str3 );

  /* concatenates str1 and str2 */
  strcat( str1, str2);
  printf("strcat( str1, str2):   %s\n", str1 );

  /* total lenghth of str1 after concatenation */
  len = strlen(str1);
  printf("strlen(str1) :  %d\n", len );

  return 0;
}
```

Output:
strcpy(str3, str1) : Hello
strcat(str1, str2): HelloWorld
strlen(str1) : 10

```
/* C Program to Count Vowels and Consonants in a String */
#include <stdio.h>
int main()
{
        char str[100];
        int i, vowels, consonants;
        i = vowels = consonants = 0;

        printf("\n Please Enter any String :  ");
        gets(str);

        while (str[i] != '\0')
        {
                if(str[i] == 'a' || str[i] == 'e' || str[i] == 'i' || str[i] == 'o' || str[i] == 'u' ||
                str[i] == 'A' || str[i] == 'E' || str[i] == 'I' || str[i] == 'O' || str[i] == 'U')
                {
                        vowels++;
                }
                else
                consonants++;
        i++;
        }
  printf("\n Number of Vowels in this String = %d", vowels);
        printf("\n Number of Consonants in this String = %d", consonants);

        return 0;
}
```

```
Please Enter any String :  Hello World

Number of Vowels in this String = 3
Number of Consonants in this String = 8
```

```
//Program to reverse a String
#include <stdio.h>
int main()
{
 char str[1000], rev[1000];
 int i, j, count = 0;
 scanf("%s", str);
 printf("\nString Before Reverse: %s", str);
 //finding the length of the string
 while (str[count] != '\0')
 {
  count++;
 }
 j = count - 1;
 //reversing the string by swapping
 for (i = 0; i < count; i++)
 {
  rev[i] = str[j];
  j--;
 }

 printf("\nString After Reverse: %s", rev);
}
```

OUTPUT:

```
Hello


String Before Reverse: Hello

String After Reverse: olleH
```

4.5 Structure

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. The, struct keyword is used to define the structure. Let's see the syntax to define the structure in c.

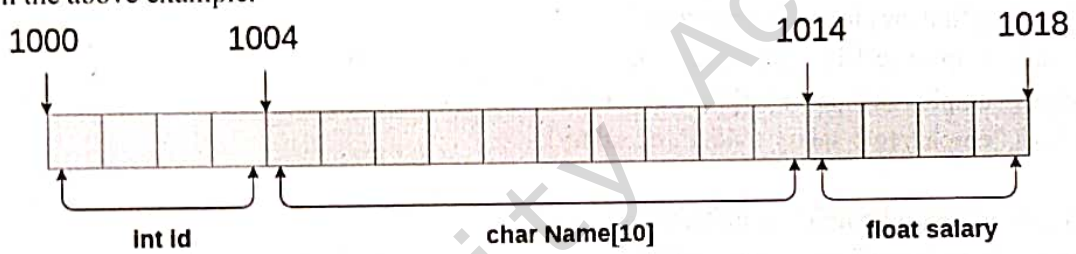
Syntax:

```
struct structure_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

Example

```
struct employee
{  int id;
   char name[10];
   float salary;
};
```

The following image shows the memory allocation of the structure employee that is defined in the above example.




```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte

Declaring structure variable

- By struct keyword within main() function
- By declaring a variable at the time of defining the structure.

1st way:

```
struct employee
{  int id;
   char name[50];
   float salary;
};
```

Now write given code inside the main() function.

```
struct employee e1, e2;
```

2nd way:

```
struct employee
{  int id;
   char name[50];
   float salary;
}e1,e2;
```

//Program: Write a C program to read and print employee details using structure

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;  //declaring e1 and e2 variables for structure
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sandeep");//copying string into char array
   e1.salary=156000;

  //store second employee information
   e2.id=102;
   strcpy(e2.name, "Ravi");
   e2.salary=126000;

   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
   printf( "employee 1 salary : %f\n", e1.salary);

   //printing second employee information
   printf( "employee 2 id : %d\n", e2.id);
   printf( "employee 2 name : %s\n", e2.name);
   printf( "employee 2 salary : %f\n", e2.salary);
   return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sandeep
employee 1 salary : 156000.000000
employee 2 id : 102
employee 2 name : Ravi
employee 2 salary : 126000.000000
```

//Program of an array of structures that stores information of 5 students and prints it.

```
#include<stdio.h>
#include <string.h>
struct student{
int rollno;
char name[10];
};
int main(){
int i;
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
printf("\nStudent Information List:");
for(i=0;i<5;i++){
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
  return 0;
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

Nested Structure in C

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee.

```
#include<stdio.h>
struct address
{
   char city[20];
   int pin;
   char phone[14];
};
struct employee
{
   char name[20];
   struct address add;
};
void main ()
{
   struct employee emp;
   printf("Enter employee information?\n");
   scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
   printf("Printing the employee information....\n");
   printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s", emp.name,emp.add.city,emp.add.pin,emp.add.phone);
}
```

Output

```
Enter employee information?
Arun
Delhi
110001
1234567890

Printing the employee information....
name: Arun
City: Delhi
Pincode: 110001
Phone: 1234567890
```

4.6 Union

Union in c language is a user-defined data type that is used to store the different type of elements.

At once, only one member of the union can occupy the memory. In other words, we can say that the size of the union in any instance is equal to the size of its largest element.

Structure	Union

struct Employee{
char x; // size 1 byte
int y; //size 2 byte
float z; //size 4 byte
}e1; //size of e1 = 7 byte
	union Employee{
	char x; // size 1 byte
	int y; //size 2 byte
	float z; //size 4 byte
	}e1; //size of e1 = 4 byte

size of e1= 1 + 2 + 4 = 7	size of e1= 4 (maximum size of 1 element)

Defining union

The union keyword is used to define the union. Let's see the syntax to define union in c.

```
union union_name
{
  data_type member1;
  data_type member2;
  .
  .
  data_type memeberN;
};
```

Example

```
union employee
{  int id;
   char name[50];
   float salary;
};
```

Program: Simple example of union in C language.

```
#include <stdio.h>
#include <string.h>
union employee
{  int id;
   char name[50];
}e1;  //declaring e1 variable for union
int main( )
{
  //store first employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  return 0;
}
```

Difference between Structure and Union

Structure	Union
You can use a struct keyword to define a structure.	You can use a union keyword to define a union.
Every member within structure is assigned a unique memory location.	In union, a memory location is shared by all the data members.
Changing the value of one data member will not affect other data members in structure.	Changing the value of one data member will change the value of other data members in union.
It enables you to initialize several members at once.	It enables you to initialize only the first member of union.
The total size of the structure is the sum of the size of every data member.	The total size of the union is the size of the largest data member.
It is mainly used for storing various data types.	It is mainly used for storing one of the many data types that are available.
It occupies space for each and every member written in inner parameters.	It occupies space for a member having the highest size written in inner parameters.
You can retrieve any member at a time.	You can access one member at a time in the union.
It supports flexible array.	It does not support a flexible array.

4.7 Enumerated Data Types

In C Programming an enumeration type (also called enum) is a user-defined data type consist of integer value and it provide meaningfull name to these value.

Syntax:

```
enum flag { integer_const1, integer_const2, ..... integer_constN };
```

the default value of integer_const1 is 0
integer_const2 is 1
.
.
integer_constN is N-1

Example:

```
#include<stdio.h>
enum week { Mon, Tue, Wed, Thu, Fri, Sat, Sun}
void main()
 {
 enum week day;
 day = wed;
 printf("%d", day);
 }
```

output

```
2
```

Example:

```
#include<stdio.h>
enum year{ Jan=10, Feb, Mar, Apr, May, Jun, Jul, Aug, Seb, Oct, Nov, Dec}
void main()
 {
 int i;
for(i=Jan; i<=Dec; i++)
  {
   printf("%d ", i)
}
```

output

```
10 11 12 13 14 15 16 17 18 19 20 21
```

Important Questions

1. What is an array? In which situation array is advantageous over linked list?
2. Write an algorithm to find second largest element in an array.
3. Differentiate structure with union.
4. Create a suitable structure in C language for keeping the records of the employees of an organization about their code, Name, Designation, Salary, Department, City of posting. Also write a program in C to enter the records of 100 employees and displays the name of those who earn more than 20,000.
5. Write a program in C to input two 3x3 matrix from the user and print multiplication as the result in matrix form.
6. Write a program that prints the real roots of a quadratic equation. Also draw flowchart for the same.
7. What do you mean by sorting? Write a program in C to sort "n" positive integers using bubble sort. Also draw the flow chart for the same.
8. Explain linear search and binary search technique for searching an item in a given array. Also write the complexity for each searching technique
9. Explain Selection sort technique for sorting problem. Also write an algorithm for selection sort. Sort the following numbers using selection sort technique. 26,54,93,17,77,31,44,55,20
10. What do you mean by order of complexity? Explain various notions to represent order of complexity with diagram.

Unit -V
Pointer & File Handling

5.1 Pointer:

C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

```
#include <stdio.h>
int main () {
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );
   return 0;
}
```

Output:
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address

type *var-name;

here type is the pointer's base type
The asterisk * used to declare a pointer
var-name is the name of the pointer variable

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

5.1.1 How to Use Pointers?

(a) We define a pointer variable,
(b) assign the address of a variable to a pointer and
(c) finally access the value at the address available in the pointer variable.

```
#include <stdio.h>
int main () {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );
   printf("Address stored in ip variable: %x\n", ip );
   printf("Value of *ip variable: %d\n", *ip );
   return 0;
}
```

Output
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20

5.1.2 NULL Pointers: It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

```
#include <stdio.h>
int main () {
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result −
The value of ptr is 0

Example-1:

```
#include <stdio.h>
int main()
{
   int* pc, c;
    c = 22;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c);
   pc = &c;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc);
    c = 11;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); *pc = 2;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c); // 2
   return 0;
}
```

Output

```
of c: 0x7fff47e7a77c
Value of c: 22
Address of pointer pc: 0x7fff47e7a77c
Content of pointer pc: 22
Address of pointer pc: 0x7fff47e7a77c

Content of pointer pc: 11

Address of c: 0x7fff47e7a77c

Value of c: 2
```

5.1.3 Pointer Arithmetic

All types of arithmetic operations are not possible with pointers. The only valid operations that can be performed are as

i. Addition of an integer to a pointer and increment operation.
ii. Subtraction of an integer from a pointer and decrement operation
iii. Subtraction of a pointer from another pointer of same type.

Pointer arithmetic is somewhat different from ordinary arithmetic. Here all arithmetic is performed relative to the size of base type of pointer. For example, if we have an integer pointer pi which contains address

- 1000 then on incrementing we get 1002 instead of 1001. This is because the size of int data type is 2.
- Similarly on decrementing pi, we will get 998 instead of 999.
- The expression (pi+3) will represent the address 1006.

Examples

int a = 5, *p = &a;
float b = 2.2, *pf = &b;
char c = 'x', *pc= &c;
Suppose the address of variables a, b and c are 1000, 4000 ,5000 respectively, so initially values of
pI, p2, p3 will be 1000, 4000 and 5000.

```
pi++;       or      ++pi;   pi =1000 + 2 = 1002 (Since int is of 2 bytes)
pi = pi-3;                  pi =1002 – 3*2 = 996
pi = pi+5;                  pi =996 + 5*2 = 1006
pi- -;      or      - -pi;  pi =1006 - 2 = 1004

pf++;       or      ++pf;   pf = 4000 + 4 = 4004 (Since float is of 4 bytes)
pf = pf-3;                  pf = 4004 – 3*4 = 3992
pf = pf+5;                  pf = 3992 + 5*4 = 4012
pf- -;      or      - -pf;  pf = 4012 - 4 = 4008

pc++;       or      ++pc;   pc = 5000 + 1= 5001 (Since char is of 1 byte)
pc = pc-3;                  pc = 5001 – 3 = 4998
pc = pc+5;                  pc = 4998 + 5 = 5003
pc- -;      or      - -pc;  pc = 5003 - 1 = 5002
```

Dynamic memory allocation in C

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

i. malloc()
ii. calloc()
iii. realloc()
iv. free()

Static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

i. C malloc()

The name "malloc" stands for memory allocation. The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
Example
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory.

ii. C calloc()

The name "calloc" stands for contiguous allocation. The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
Example:
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type float**C**

iii. free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Example 1: malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int n, i, *ptr, sum = 0;
```

```
   printf("Enter number of elements: ");
   scanf("%d", &n);
   ptr = (int*) malloc(n * sizeof(int));
    // if memory cannot be allocated
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);

   // deallocating the memory
   free(ptr);
   return 0;
}
```

```
Output:
Enter number of elements: 5
Enter elements: 4
3
5
8
9
Sum = 29
```

Example 2: calloc() and free()

```
// Program to calculate the sum of n numbers entered by the user
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int n, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   ptr = (int*) calloc(n, sizeof(int));
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
```

```
Output:
Enter number of elements: 5
Enter elements: 4
5
8
9
2
Sum = 28
```

```
    free(ptr);
    return 0;
}
```

iv. C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, ptr is reallocated with a new size x.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
   printf("Addresses of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
        printf("%u\n",ptr + i);
    printf("\nEnter the new size: ");
    scanf("%d", &n2);
    // rellocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));
    printf("Addresses of newly allocated memory: ");
    for(i = 0; i < n2; ++i)
        printf("%u\n", ptr + i);
     free(ptr);
    return 0;
}
```

Output

```
Enter size: 2
Addresses of previously allocated
memory:26855472
26855476

Enter the new size: 4
Addresses of newly allocated
memory:26855472
26855476
26855480
26855484
```

5.2 Use of pointers in self-referential structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

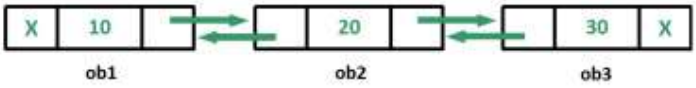
```
struct node {
   int data1;
   char data2;
   struct node* link;
};

int main()
{
   struct node ob;
   return 0;
}
```

In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

5.2.1 Types of Self Referential Structures

i. Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.

ii. Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.



```
//Program to understand Self Referential Structures
#include <stdio.h>
struct node {
int data1;
char data2;
struct node* link;
};
int main()
{
struct node ob1; // Node1
// Initialization
ob1.link = NULL;
ob1.data1 = 10;
ob1.data2 = 20;
struct node ob2; // Node2
// Initialization
ob2.link = NULL;
ob2.data1 = 30;
ob2.data2 = 40;
// Linking ob1 and ob2
ob1.link = &ob2;
// Accessing data members of ob2 using ob1
printf("%d", ob1.link->data1);
printf("\n%d", ob1.link->data2);
return 0;
}
```

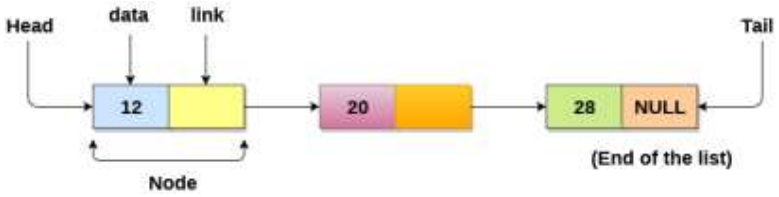
Output:

30

40

5.2.2 Linked List

Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory. The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

- The size of array must be known in advance before using it in the program.
- Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

5.3. File management

A File can be used to store a large volume of persistent data. Like many other languages 'C' provides following file management functions,

i. Creating a new file
ii. Opening an existing file
iii. Closing a file
iv. Reading from and writing information to a file

functions available in 'C

function	purpose
fopen ()	Creating a file or opening an existing file
fclose ()	Closing a file
fprintf ()	Writing a block of data to a file
fscanf ()	Reading a block data from a file
getc ()	Reads a single character from a file
putc ()	Writes a single character to a file
getw ()	Reads an integer from a file
putw ()	Writing an integer to a file
fseek ()	Sets the position of a file pointer to a specified location
ftell ()	Returns the current position of a file pointer
rewind ()	Sets the file pointer at the beginning of a file

5.3.1 Creating or opening file using fopen()

The fopen() function is used to create a new file or open an existing file in C.The fopen function is defined in the stdio.h header file.

Now, lets see the syntax for creation of a new file or opening a file
FILE *ptr;
ptr = fopen(“file_name”, “mode”)

Eg.
ptr=fopen("E:\\cprogram\\newprogram.txt","w");
ptr=fopen("E:\\cprogram\\oldprogram.bin","rb");

Opening Modes in Standard I/O		
Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
ab	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

5.3.2 Closing a File

The file (both text and binary) should be closed after reading/writing. Closing a file is performed using the fclose() function.

```
fclose(ptr);
```

Here, ptr is a file pointer associated with the file to be closed.

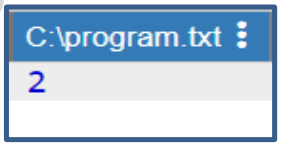
5.3.3 Reading and writing to a text file For reading and writing to a text file, we use the functions fprintf() and fscanf(). They are just the file versions of printf() and scanf(). The only difference is that fprint() and fscanf() expects a pointer to the structure FILE.

Example 1: Write to a text file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int num;
  FILE *fptr;
  // use appropriate location if you are using MacOS or Linux
  fptr = fopen("C:\\program.txt","w");
  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }
  printf("Enter num: ");
  scanf("%d",&num);
  fprintf(fptr,"%d",num);
  fclose(fptr);
  return 0;
}
```

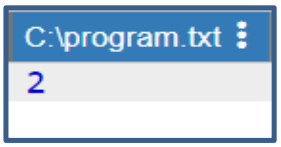
Output

Enter num: 2



Example 2: Read from a text file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.txt","r")) == NULL){
      printf("Error! opening file");

      // Program exits if the file pointer returns NULL.
      exit(1);
  }
  fscanf(fptr,"%d", &num);
  printf("Value of n=%d", num);
  fclose(fptr);
   return 0;
}
```



Output

Value of n=2

/*Write a C program to read numbers from a file and write even, odd and prime numbers in separate files.*/

```
#include <stdio.h>
 void main() {
      FILE *fp1, *fp2, *fp3, *fp4;
      int n, i, num, flag = 0;

      /* open data.txt in read mode */
      fp1 = fopen("C:\\data.txt", "w");
      printf("Enter the value for n:");
      scanf("%d", &n);
      for (i = 0; i <= n; i++)
            fprintf(fp1, "%d ", i);
      fprintf(fp1, "\n");
      fclose(fp1);

      /* open files to write even, odd and prime nos separately */
      fp1 = fopen("C:\\data.txt", "r");
      fp2 = fopen("C:\\even.txt", "w");
      fp3 = fopen("C:\\odd.txt", "w");
      fp4 = fopen("C:\\prime.txt", "w");

      fprintf(fp2, "Even Numbers:\n");
      fprintf(fp3, "Odd Numbers:\n");
      fprintf(fp4, "Prime Numbers:\n");

      /* print even, odd and prime numbers in separate files */
      while (!feof(fp1)) {
            fscanf(fp1, "%d", &num);
            if (num % 2 == 0) {
                  fprintf(fp2, "%d ", num);
            } else {
                  if (num > 1) {
                        for (i = 2; i < num; i++) {
                              if (num % i == 0) {
                                    flag = 1;
                                    break;
                              }
                        }
                        if (!flag) {
                              fprintf(fp4, "%d ", num);
                        }
                  }
                  fprintf(fp3, "%d ", num);
```

```
                flag = 0;
            }
        }
        fprintf(fp2, "\n");
        fprintf(fp3, "\n");
        fprintf(fp4, "\n");

        /* close all opened files */
        fclose(fp1);
        fclose(fp2);
        fclose(fp3);
        fclose(fp4);
}
```

Output

```
Enter the value for n:20
```

main.c | C:\data.txt | C:\even.txt | C:\odd.txt | C:\prime.txt

C:\data.txt

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

C:\even.txt

```
Even Numbers:
0
2
4
6
8
10
12
14
16
18
20
20
```

C:\odd.txt

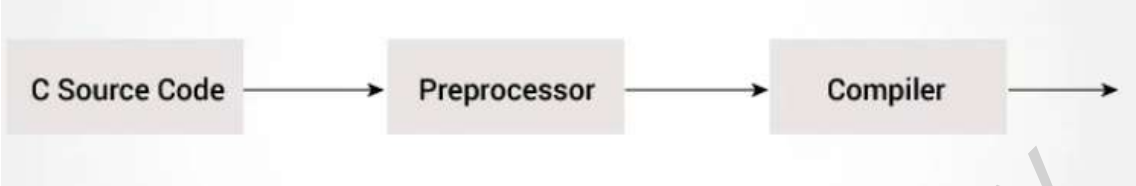
```
Odd Numbers:
1
3
5
7
9
11
13
15
17
19
```

C:\prime.txt

```
Prime Numbers:
3
5
7
11
13
17
19
```

5.4 C Preprocessor and Macros

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. All preprocessor commands begin with a hash symbol (#).



5.4.1 Important preprocessor directives

Sr.No.	Directive & Description	
i.	**#define**	Substitutes a preprocessor macro.
ii.	**#include**	Inserts a particular header from another file.
iii.	**#undef**	Undefines a preprocessor macro.
iv.	**#ifdef**	Returns true if this macro is defined.
v.	**#ifndef**	Returns true if this macro is not defined.
vi.	**#if**	Tests if a compile time condition is true.
vii.	**#else**	The alternative for #if.
viii.	**#elif**	#else and #if in one statement.
ix.	**#endif**	Ends preprocessor conditional.
x.	**#error**	Prints error message on stderr.

Macros using #define : A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive.

Example 1: #define preprocessor

```
#include <stdio.h>
#define PI 3.1415
int main()
{
   float radius, area;
   printf("Enter the radius: ");
   scanf("%f", &radius);
   // Notice, the use of PI
   area = PI*radius*radius;
   printf("Area=%.2f",area);
   return 0;
}
```

Inserts a particular header using #include: These directives tell the CPP to get *stdio.h* from System Libraries and add the text to the current source file. The next line tells CPP to get *myheader.h* from the local directory and add the content to the current source file.

```
#include <stdio.h>
#include "myheader.h"

// Creating your own header file and using it accordingly.
```

```
void add(int a, int b)
{
   printf("Added value=%d\n", a + b);
}
void multiply(int a, int b)
{
   printf("Multiplied value=%d\n", a *
b);
}
```

myhead.h

```
#include <stdio.h>
#include "myhead.h"
int main()
{
   add(4, 6);
     /*This calls add function written in myhead.h
     and therefore no compilation error.*/
   multiply(5, 5);

   // Same for the multiply function in myhead.h
   printf("BYE!See you Soon");
   return 0;
}
```

arithmetic.c

Output:

```
Added value:10
Multiplied value:25
BYE!See you Soon
```

#undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

```
#include <stdio.h>
#define PI 3.14
#undef PI
main() {
  printf("%f",PI);
}
```

Output:

Compile Time Error: 'PI' undeclared

```
#include <stdio.h>
#define number 15
int square=number*number;
#undef number
main() {
  printf("%d",square);
}
```

Output:

225

#ifdef , #else and #endif : The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present. The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

```
#include <stdio.h>
#include <conio.h>
#define NOINPUT
void main() {
int a=0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

2

```
#include <stdio.h>
#include <conio.h>
void main() {
int a=0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif

printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Enter a:5
Value of a: 5
```

#ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

```
#include <stdio.h>
#include <conio.h>
#define INPUT
void main() {
int a=0;
#ifndef INPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Enter a:5
Value of a: 5
```

```
#include <stdio.h>
#include <conio.h>
void main() {
int a=0;
#ifndef INPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Value of a: 2
```

#if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 0
void main() {
#if (NUMBER==0)
printf("Value of Number is: %d",NUMBER);
#endif
getch();
}
```

Output:

```
Value of Number is: 0
```

#error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

```
#include<stdio.h>
#ifndef __MATH_H
#error First include then compile
#else
void main(){
   float a;
   a=sqrt(7);
   printf("%f",a);
}
#endif
```

Output:
Compile Time Error: First include then compile

5.5 Command Line Argument in C

Command line argument is a parameter supplied to the program when it is invoked. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

int main(int argc, char *argv[])

main() function of a C program accepts arguments from command line or from other shell scripts by following commands. They are,

argc – Number of arguments in the command line including program name

argv[] – This is carrying all the arguments

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])   //  command line arguments
{
if(argc!=5)
{
  printf("Arguments passed through command line not equal to 5");
  return 1;
}
   printf("\n Program name  : %s \n", argv[0]);
  printf("1st arg  : %s \n", argv[1]);
  printf("2nd arg  : %s \n", argv[2]);
  printf("3rd arg  : %s \n", argv[3]);
  printf("4th arg  : %s \n", argv[4]);
  printf("5th arg  : %s \n", argv[5]);

return 0;
}
```

Save this program as test.c
we run the executable “test” along with 4 arguments in command line like below.

```
 $./a.out this is a program
```

OUTPUT:

```
Program name : test
1st arg : this
2nd arg : is
3rd arg : a
4th arg : program
5th arg : (null)
```

Important Questions

1. What do you mean by pointer arithmetic?
2. What is dynamic memory allocation? Explain the calloc(), malloc(), ealloc() and free() functions in detail. What is lifetime of a variable,which is created dynamically?
3. Explain the importance of pointers in C. Write a program in C to swap the values of two numbers entered by user using function call by reference method
4. Write macro definition with arguments for calculation of simple interest and amount. Store these macro definitions in a file called 'interest.h'. Include this file in your program and use the macro definitions for calculating simple interest and amount.
5. What are different file opening modes? Write a program in C that reads a series of integer numbers from a file named INPUT and write all odd numbers to a file to be called ODD and all even numbers to a file to be called EVEN.
6. Write a short note on following pre-processor directives with example:
    a. Macro Expansion
    b. File Inclusion
7. Explain command line arguments in C with the help of example.
8. Define various file operations in C. Write a program in C to count and print the number of characters in a file.